Elements of Web Server and concurrency (in go)

30 June 2016

Valentin Deleplace
Developer

Warning

Some code will be displayed.

Cross-compilation

Cross-compilation

This is the compilation script of one of our tools **patapon**, run from my linux workstation:

```
env GOOS=darwin  GOARCH=amd64 go build -o target/osx64/patapon
env GOOS=windows GOARCH=amd64 go build -o target/win64/patapon.exe
env GOOS=linux   GOARCH=amd64 go build -o target/linux64/patapon
```

That's it.

To ship, copy the 3 executables to shared folder.

Executables are statically linked.

First-class functions, and closures

Simple chronometer

```
t := time.Now()

fetchData()

duration := time.Since(t)
```

Simple chronometer: with func argument

```
duration := clock(fetchData)
```

```
func clock(f func()) time.Duration {
    t := time.Now()
    f()
    return time.Since(t)
}
```

Chronometer: time a block of code?

```
t := time.Now()

for _, r := range resources {
    fetch(r)
}

duration := time.Since(t)
```

Wrap code in a closure

```
duration := clock(func() {
    for _, r := range resources {
        fetch(r)
    }
})
```

Fork/Join

Utility goroutine spawner

```go
// RunConcurrent launches funcs,
// and waits for their completion.
func RunConcurrent(funcs ...func()) {
	var wg sync.WaitGroup
	wg.Add(len(funcs))
	for _, f := range funcs {
		f := f
		go func() {
			f()
			wg.Done()
		}()
	}
	wg.Wait()
}
```

You may often write such code to fit your needs.

Utility goroutine spawner

```go
// RunConcurrent launches funcs,
// and waits for their completion.
func RunConcurrent(funcs ...func()) {
	var wg sync.WaitGroup
	wg.Add(len(funcs))
	for _, f := range funcs {
		f := f
		go func() {
			f()
			wg.Done()
		}()
	}
	wg.Wait()
}
```

Func arguments (variadic)

Utility goroutine spawner

```go
// RunConcurrent launches funcs,
// and waits for their completion.
func RunConcurrent(funcs ...func()) {
	var wg sync.WaitGroup
	wg.Add(len(funcs))
	for _, f := range funcs {
		f := f
		go func() {
			f()
			wg.Done()
		}()
	}
	wg.Wait()
}
```

Use a `sync.WaitGroup` to join at completion

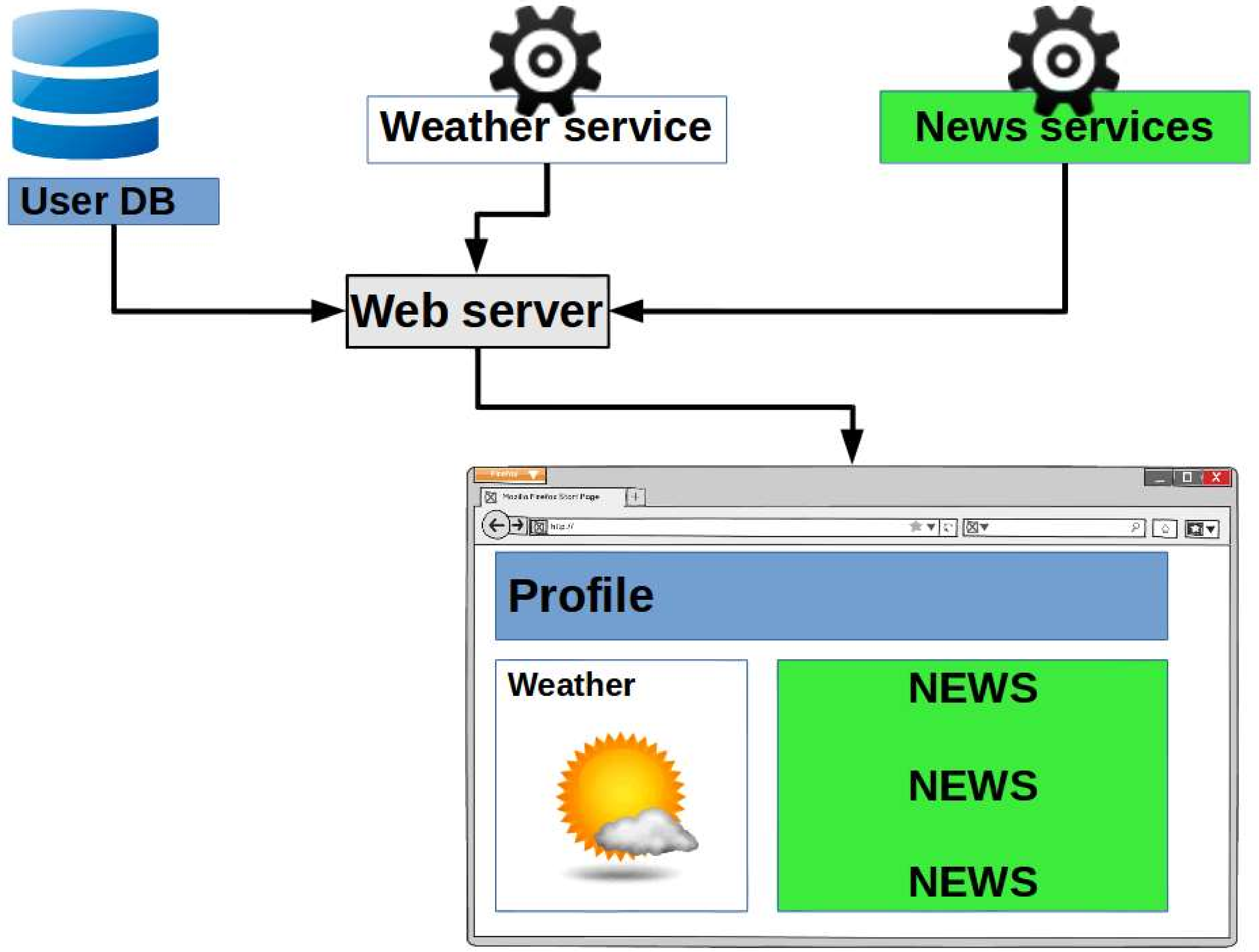
Important: call `wg.Add` *before* spawning the goroutines

Utility goroutine spawner

```
// RunConcurrent launches funcs,
// and waits for their completion.
func RunConcurrent(funcs ...func()) {
    var wg sync.WaitGroup
    wg.Add(len(funcs))
    for _, f := range funcs {
        f := f
        go func() {
            f()
            wg.Done()
        }()
    }
    wg.Wait()
}
```

Must bind to a new variable at each loop iteration

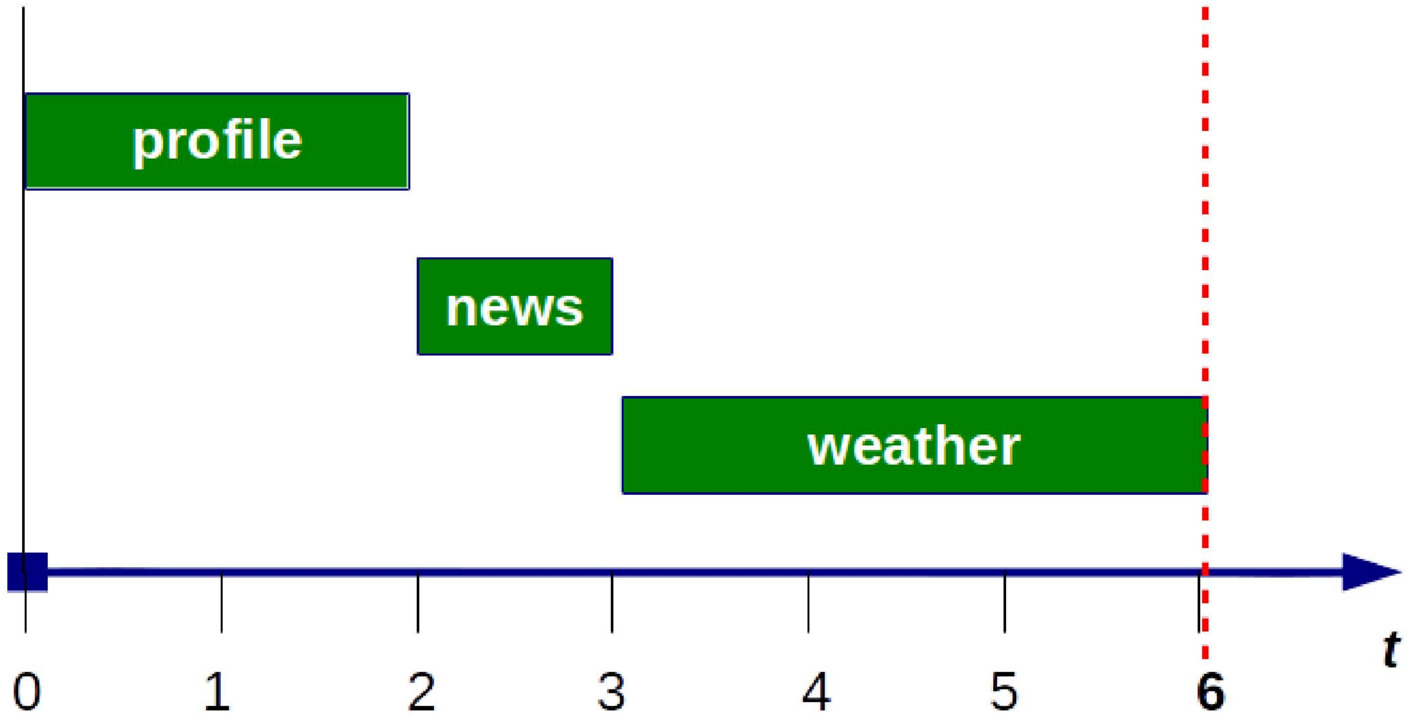
Example: page blocks from data sources



Serial exec

```
    profile()
    news()
    weather()
```

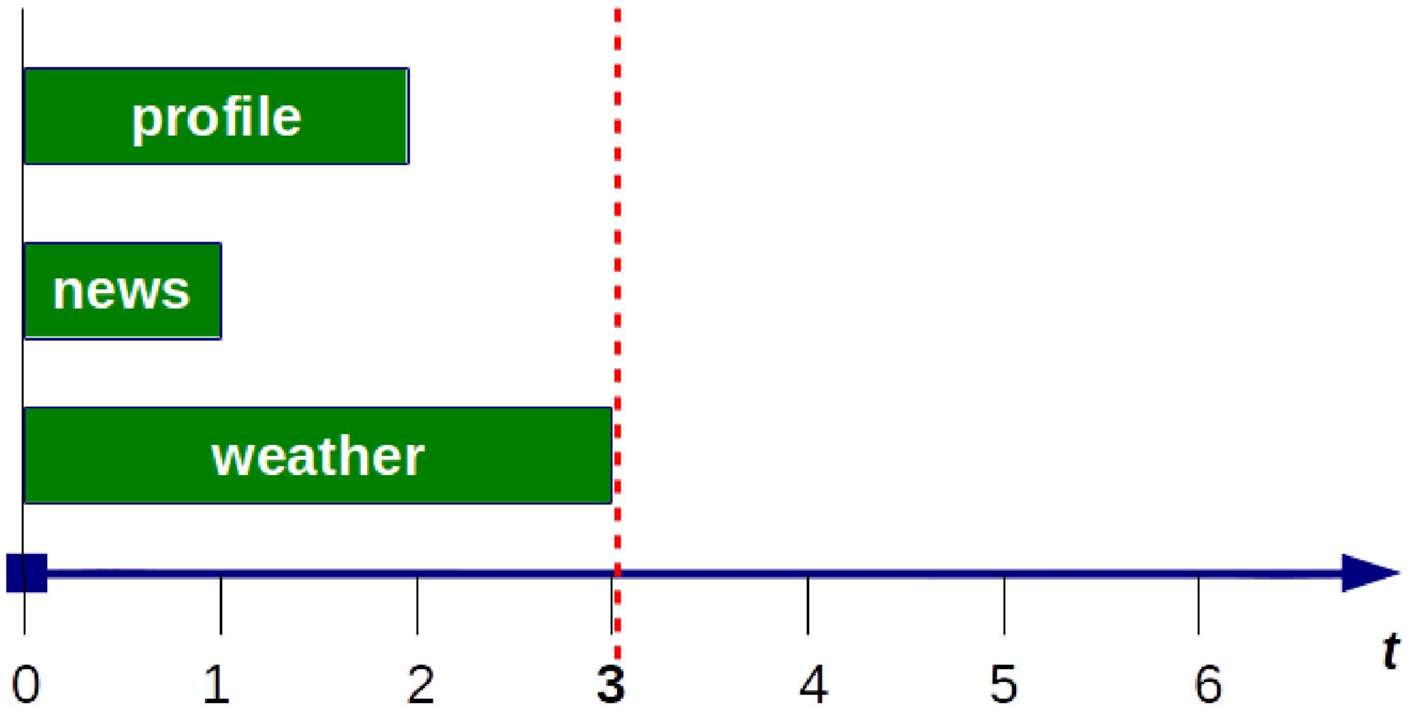
Run



Concurrent exec

```
    RunConcurrent(
        profile,
        news,
        weather,
    )
```

Run



It is straightforward and convenient, but be very careful with concurrency!

What if my services don't have the same signature?

```
func profile(username string) error {...}

func news() {...}

func weather(city string, day time.Time) {...}
```

Wrap in closures

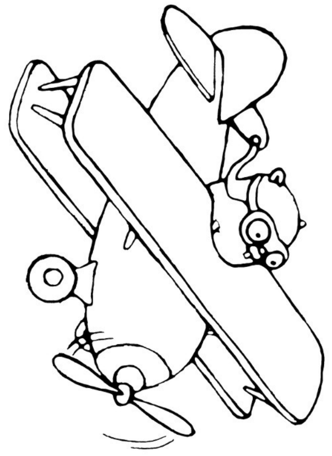
```
RunConcurrent(
    func() { perr = profile(currentuser) },
    news,
    func() { weather(city, time.Now()) },
)
```

- Closures can **read** variables from outside their body
- Closures can **write** variables from outside their body

Warning

Concurrency and parallelism are **hard** and **subtle**.

They are made easier in go, but there is still plenty of room to shoot self in the foot.



Testing and data races

A simple server : hit counter

```go
package main

import "net/http"

func main() {
    http.HandleFunc("/", h)
    http.ListenAndServe(":7070", nil)
}

var count int = 0

func h(w http.ResponseWriter, r *http.Request) {
    count++
}
```

Test the handler concurrently

```
func TestHandler(t *testing.T) {
    count = 0
    for i := 0; i < 200; i++ {
        go h(nil, nil)
    }

    time.Sleep(3 * time.Second)
    if count != 200 {
        t.Errorf("Expected %d hits, got %d", 200, count)
    }
}
```

```
go test
PASS
ok
```

```
go test
--- FAIL: TestHandler (3.00s)
    racy_handler_test.go:16: Expected 200 hits, got 194
FAIL
```

Use the built-in race detector

```
go test -race

==================
WARNING: DATA RACE
Read by goroutine 8:
  racy_server.go:13 +0x30

Previous write by goroutine 7:
  racy_server.go:13 +0x4c

Goroutine 8 (running) created at:
  racy_handler_test.go:12 +0x86
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:473 +0xdc

Goroutine 7 (finished) created at:
  racy_handler_test.go:12 +0x86
  testing.tRunner()
      /usr/local/go/src/testing/testing.go:473 +0xdc
==================
--- FAIL: TestHandler (3.01s)
racy_handler_test.go:16: Expected 200 hits, got 162
FAIL
```

Test the server concurrently

```
func TestServer(t *testing.T) {
    count = 0
    go main()
    for i := 0; i < 200; i++ {
        go http.Get("http://localhost:7070/")
    }

    time.Sleep(3 * time.Second)
    if count != 200 {
        t.Errorf("Expected %d hits, got %d", 200, count)
    }
}
```

Use -race in production

- if you have many servers, enable race detector on one of them
- or enable it on your server for a few hours

Overhead : measure the performance penalty. It might be acceptable (e.g. +100%, +200%). Find as many concurrency bugs as possible.

Pprof

Pprof

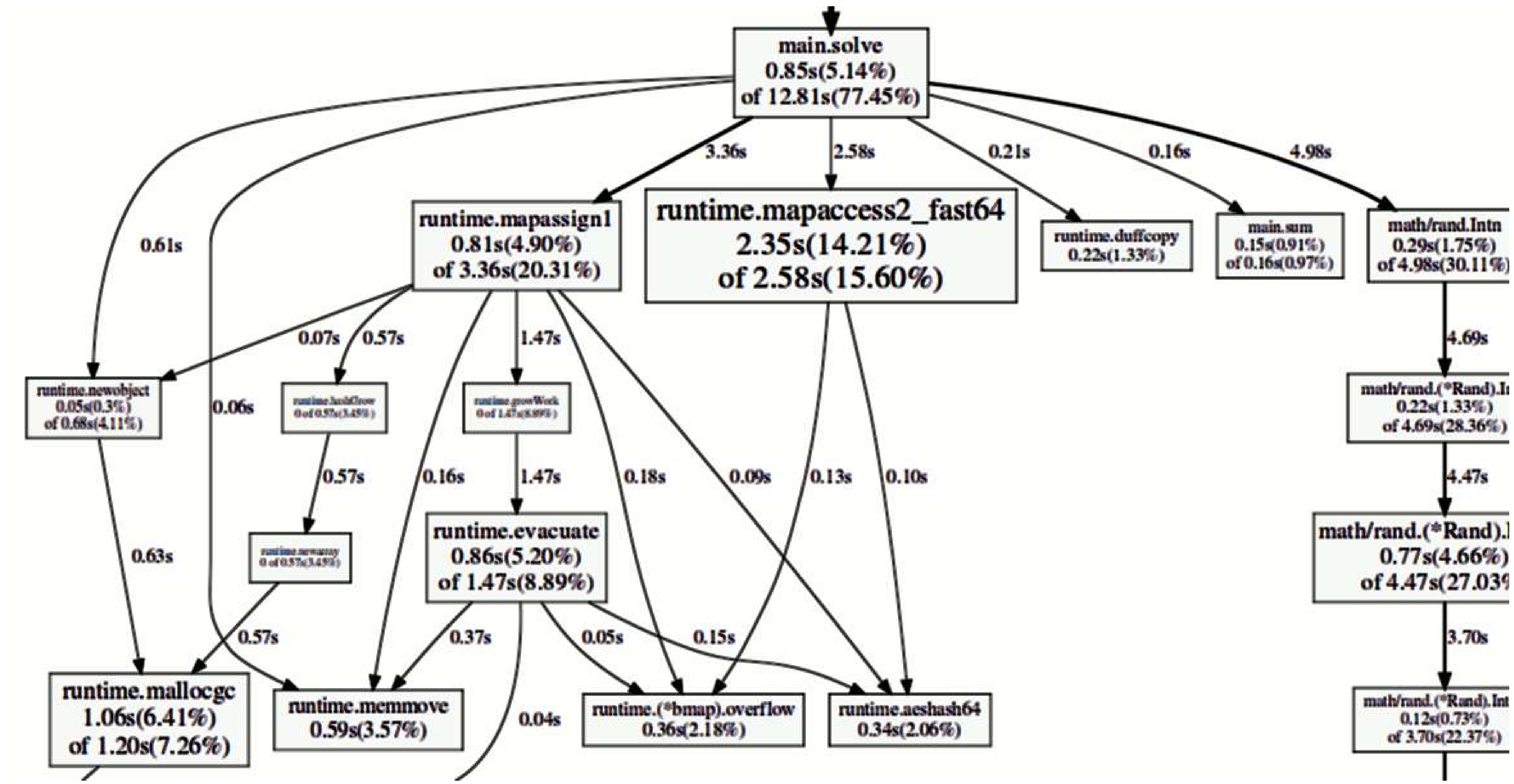
- CPU profiler
- Memory profiler
- Find and visualize bottlenecks

Easy setup:

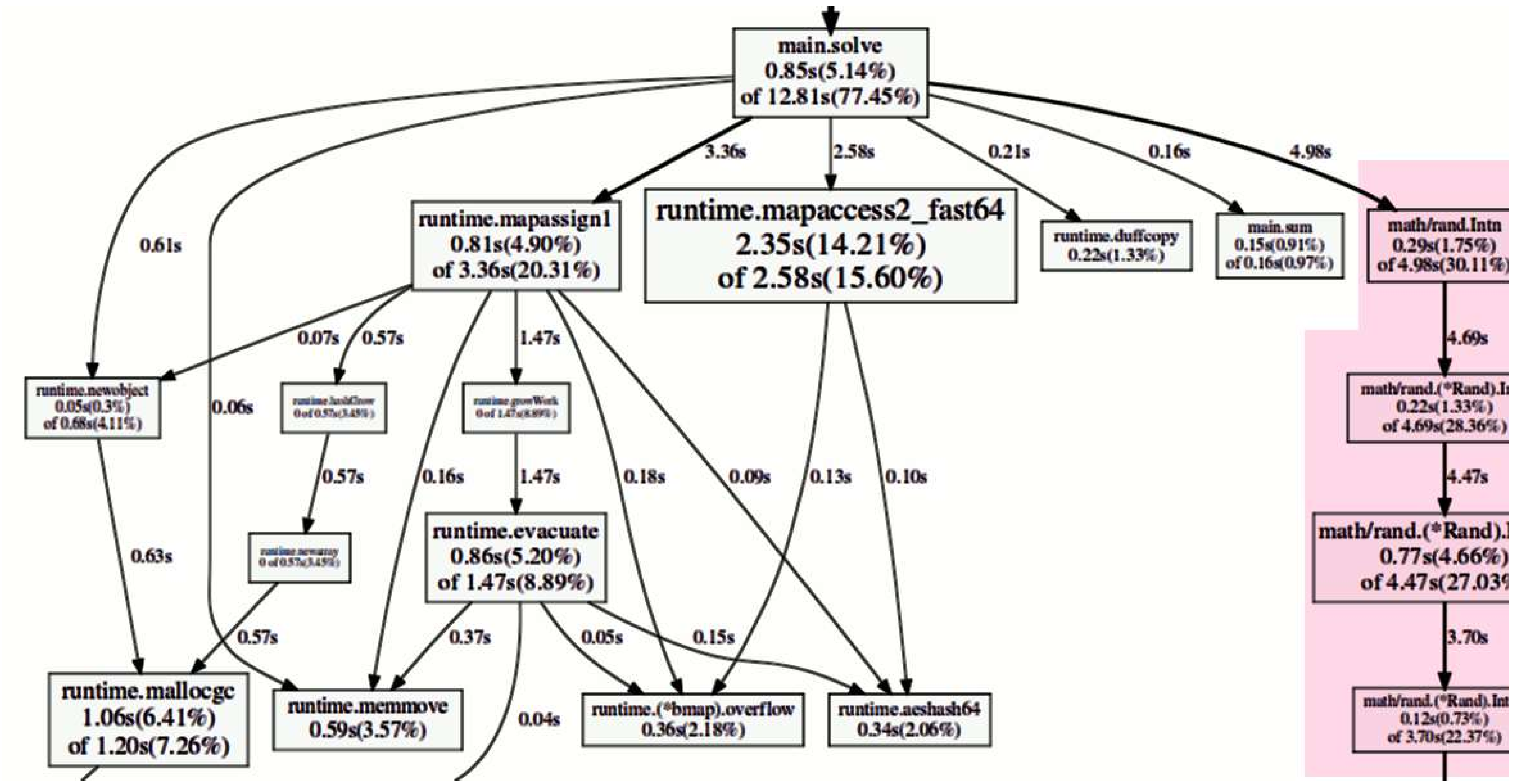
```
import "github.com/pkg/profile"

func main() {
    defer profile.Start().Stop()
    ...
}
```

Then inspect the profile data (it's a file).

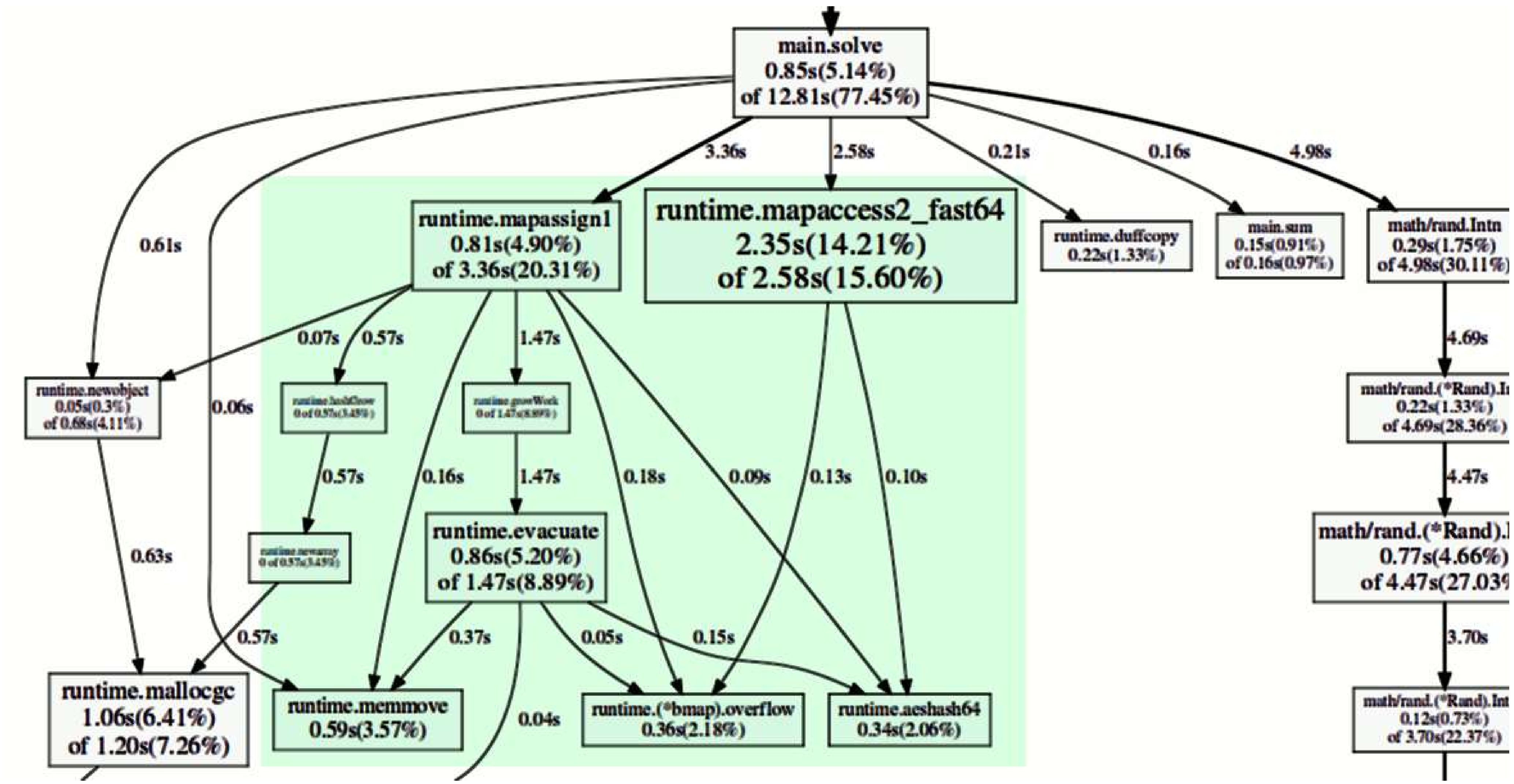
Pprof CPU profile



Pprof CPU profile



Pprof CPU profile



Links : standard library

Package **net/http** (https://golang.org/pkg/net/http/)

very good for a wide range of HTTP server needs

Package **sync** (https://golang.org/pkg/sync/)

includes WaitGroup, Mutex, etc.

Package **text/template** (https://golang.org/pkg/text/template/)

Package **html/template** (https://golang.org/pkg/html/template/)

efficient (compiled) and safe templating.

Links : misc

Go-Traps (http://go-traps.appspot.com/)

Benign data races: what could possibly go wrong? (https://software.intel.com/en-us/blogs/2013/01/06/benign-data-races-what-could-possibly-go-wrong)

Introducing the Go Race Detector (https://blog.golang.org/race-detector)

Advanced Go Concurrency Patterns (https://talks.golang.org/2013/advconc.slide)

High Performance Apps with Go on App Engine (https://talks.golang.org/2013/highperf.slide)

Program your next server in Go (https://talks.golang.org/2016/applicative.slide)

Thank you

Valentin Deleplace
Developer