

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Providing timing guarantees in software using Golang

Permalink

<https://escholarship.org/uc/item/7q53h0p6>

Author

Kashinath, Ashish

Publication Date

2017

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Providing timing guarantees in software using Golang

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Ashish Kashinath

Committee in charge:

Professor Rajesh K. Gupta, Chair
Professor Aaron D Shalev
Professor Deian Stefan

2017

Copyright
Ashish Kashinath, 2017
All rights reserved.

The thesis of Ashish Kashinath is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2017

DEDICATION



My grandfather, Late Mr. V.U.Lembodaran

I remember the old white car we drove,
I remember us walking through the narrow streets,
You in front, me right behind,
To buy the best things for me to take back home,

I remember all the good times we had,
The morning walk to temples, the stop at the flower-shop,
And the numerous five-rupee coins you gave me,
To seek the blessings of the Almighty,

I remember the countless people you introduced to me,
For almost everyone knew you on the streets,
As we were getting the daily newspaper on our way back,
You always had a genuine smile for everyone,
And would always stop to enquire their well-being,

I remember our countless visits to restaurants,
And all the dosas, vadas you bought for me,
Only for you to see me eat them,
While you slowly drank coffee, sipping on it from the saucer,

I remember how you sat in your chair,
Injured physically but quietly hopeful on the inside,
As I walked away to chase my dreams,

I will forever regret not seeing you again,
And would give anything to have you back again,
For nothing in my life feels the same without you,
And every achievement without you unforgivably pyrrhic,

My prayers were not enough,
As you drifted away from me slowly,
But I know you are there with me in spirit,
Quietly walking ahead of me like we always used to.

This thesis is dedicated to the loving memory of my grandfather, Late Mr. Lembodaran who was humble, unassuming, and caring towards everyone around.

EPIGRAPH



THE AUTHOR OF THE WINDOWS FILE
COPY DIALOG VISITS SOME FRIENDS.

TABLE OF CONTENTS

Signature Page	iii
Dedication	iv
Epigraph	v
Table of Contents	vii
List of Figures	ix
List of Tables	xi
Acknowledgements	xii
Abstract of the Thesis	xiii
Chapter 1 Introduction	1
1.1 Runtime Environments	2
1.2 Writing Realtime Software	3
1.3 Reason for choosing Golang for this work	4
1.3.1 Issues with Current Golang for Realtime Programming	5
1.4 Problem statement	6
1.5 Contributions	6
1.6 Layout of Thesis	6
Bibliography	7
Chapter 2 Background	8
2.1 Introduction to Golang	8
2.2 Goals of Golang	9
2.3 Characteristics of Golang	9
2.4 Where is Golang used?	10
2.5 Brief overview of Golang	10
2.6 Background in Golang	11
2.7 Golang runtime environment	21
2.8 Benchmarking Timing of Golang Applications	21
2.9 Conclusion	31
Bibliography	33

Chapter 3	The Golang Runtime Scheduler	34
	3.1 Introduction to Scheduling	35
	3.1.1 Scheduling Terminology	35
	3.1.2 States of a Task	38
	3.1.3 Scheduling Types	39
	3.2 Scheduling in the Golang runtime environment	40
	3.2.1 Why does scheduling have to be done at the user-level runtime environment?	40
	3.2.2 Golang’s Threading Model	42
	3.3 Structures in the Golang runtime scheduler	43
	3.3.1 Initialization of the Runtime Environment and Scheduler startup	44
	3.3.2 Working details of the Runtime Scheduler	51
	3.4 Golang and Realtime Programming	55
	3.4.1 Viability of Realtime Programming Using Golang	55
	3.4.2 Study of Scheduler Latency	56
	3.5 Realtime Golang	59
	3.6 Conclusion	63
	Bibliography	65
Chapter 4	The Golang Garbage Collector (GC)	67
	4.1 Overview of Garbage Collection	67
	4.2 Garbage Collection Internals	68
	4.3 Memory Allocation in Golang runtime	69
	4.3.1 Advantages of TCMalloc over standard malloc	69
	4.3.2 Data Structures Terminology in Memory Allocation	70
	4.4 Analyzing the Garbage Collection in Golang	70
	4.4.1 Binary Tree Benchmark	71
	4.4.2 Golang Package Parser	71
	4.5 Conclusion	74
	Bibliography	75
Chapter 5	Conclusion	76

LIST OF FIGURES

Figure 2.1:	Block diagram of the runtime environment and its interactions with the OS	21
Figure 2.2:	Example execution of the <code>time</code> command for <code>go fmt</code> operation.	23
Figure 2.3:	Example execution of <code>GODEBUG</code> command to capture scheduler traces along with the timestamps.	25
Figure 2.4:	Example execution of <code>go test</code> and <code>pprof</code> tools to find functions consuming most time of the CPU. We can see that the function <code>casgstatus()</code> and <code>newproc1()</code> are consuming most CPU time here which is expected as the former is called when the status of a goroutine changes and the latter is called whenever a goroutine is created. These functions are defined in the runtime environment in the file <code>proc.go</code>	27
Figure 2.5:	Example execution of <code>png</code> command to find functions consuming most time of the CPU as well as an overview of the callgraph of the codebase. We can see that the function <code>casgstatus()</code> and <code>newproc1()</code> are consuming most CPU time here as well which is shown by the big size of the boxes representing these two functions.	28
Figure 2.6:	<code>go tool trace</code> output showing the timeline of execution of the goroutines. We see that the 10 spawned goroutines are indicated in the right bottom of the graph.[7]	32
Figure 3.1:	Example Screenshot of tasks in an Operating System.	38
Figure 3.2:	Different entities used in the Golang runtime scheduler	44
Figure 3.3:	Salient fields in the goroutine structure(<code>G</code>)	46
Figure 3.4:	Salient fields in the worker thread structure (<code>M</code>)	47
Figure 3.5:	Salient fields in the context structure (<code>P</code>)	48
Figure 3.6:	Tracing the call-graph of a goroutine as it passes from the user program to the runtime environment.	53
Figure 3.7:	Using the <code>perf</code> tool to measure the context switching time	57

Figure 4.1: Trace output of the <code>go tool trace</code> command for the Binary Tree Benchmark showing pause times of 2 milliseconds. . . .	72
Figure 4.2: Trace output of the <code>go tool trace</code> command for the Go Parser package showing variable pause times.	73

LIST OF TABLES

Table 3.1: Process State Codes used in standard Linux System	38
Table 3.2: Possible states of a goroutine and their definitions	45
Table 3.3: Possible states of a context and their implication	45
Table 3.4: Context Switching Time (Scheduling Latency): Golang runtime Vs. Linux	59

ACKNOWLEDGEMENTS

I thank my advisor Professor Rajesh Gupta for motivating me and guiding me through the project. I am grateful for all the suggestions I received on the work. Also, I am indebted to Professor Aaron Shalev and Deian Stefan for their time in meeting with me, patiently reading my manuscript, and offering their suggestions to turn this piece into a readable one.

Furthermore, I am thankful to my friends Sean Hamilton and Dhiman Sengupta for their tips and suggestions. I am also extremely thankful to my friends Francesco Fraternali and Mike Barrow for their sense of humour and for keeping the lab atmosphere very relaxed.

Finally, nothing would have been possible without the support and positive energy given to me by my grandmother Radha, father N.Kashinath, mother Latha, and sister Shruthi. This one is for you.

ABSTRACT OF THE THESIS

Providing timing guarantees in software using Golang

by

Ashish Kashinath

Master of Science in Computer Science

University of California, San Diego, 2017

Professor Rajesh K. Gupta, Chair

Guarantee of timing performance is key in realtime systems. Realtime software that does not meet timing requirements is considered erroneous regardless of its functional correctness. Ensuring timing accuracy is challenging due to differences in the notion of time between hardware, software and real-world interfaces. In practice, this challenge is addressed by separating the program development and runtime environments for realtime versus general-purpose (or non-realtime) software. Typically, realtime operating environments are characterized by simple process scheduling, minimal or no memory management, and a fixed set of processes having static priorities. Thus, realtime software development is disjoint from other forms of system software development.

This work focuses on developing an outline of a software platform for writing realtime software. This platform is aimed for use in standard operating systems and thus targets co-existence with non-realtime software. To accomplish this goal, we use the Golang programming language and its runtime environment. Golang is a type-safe and memory-safe systems programming language, designed from the ground up as a ‘C for the 21st century’. Golang offers a rich runtime environment that performs scheduling, memory allocation and cleaning and, synchronization. We present initial experimental results of using Golang for realtime applications.

Chapter 1

Introduction

Time is a central concept in cyber-physical systems. The notion of time may possibly vary among the hardware systems, the software systems, and the interfacing between the system to the real-world. The key aspect of time is its accuracy, rather than the speed of operation. Consequently, we need to ensure that the timing is guaranteed in each part of the system, and that it meets the system requirements. Real-time software is a class of software, in which a deviation in the program execution from the expected time is viewed as a failure irrespective of its correctness in terms of functionality.

Examples of realtime software include software running on pacemakers, anti-braking systems in vehicles, and avionics systems.

In current software practice, the programming and program execution are divided into two distinct phases. The first is the Compile-time environment that provides support for static checks via the type system and program analysis. The second is the Runtime environment that provides for loading of the compiled program into memory and enables its interaction with the operating system(OS). The runtime environment, enables a program written in a programming language to run on the OS by implementing an execution model. The execution model refers to the organization of the compiled program on the OS, which include facilities for memory allocation/deallocation, synchronization, and garbage collection. Thus, the runtime environment makes it easier for the running program to interact with the low-level subsystem by switching into the low-level execu-

tion model; it thus helps the OS in performing different tasks on the running program.

Examples of runtime environment include virtual machines for java bytecode such as Android Runtime, the bare CPU for native code, and the C++ runtime environment for C++ code.

1.1 Runtime Environments

Over the years, several runtime environments have been developed that provide a specialized execution environment for programs on different CPU-OS combinations. Some runtime environments focus on providing a parallel programming interface such as Cilk, which enables a multithreaded programming environment.[1] Another interesting example of a runtime environment is the hypervisor which translates a guest OS application into an underlying host OS operation. For instance, a network access on a guest OS uses a ‘virtualized network interface card’(VNIC) to communicate with the host OS, thus providing portability and a consistent interface. On the other end of the spectrum, a runtime environment can be an API such as the POSIX threads(pthread) which furnish a parallel programming model for the user.[2]

Usually, changing the low-level OS properties such as stack space for a method or changing the heap size threshold to trigger garbage collection requires recompiling the OS kernel. This slowly gets complicated since there are numerous applications, each requiring a separate view of low-level OS behaviour. Runtime environment addresses this issue by providing APIs which call into the OS execution environment. The Golang runtime environment performs multiple tasks such as memory allocation and collection, managing stacks, goroutines, channels, and reflection. With a rich set of functionalities provided by the runtime system, the Golang runtime environment is a great test bed for programming.

1.2 Writing Realtime Software

Systems programming typically requires a functionally correct piece of software, be it a user-space application intending to upload images on the internet or a kernel driver for a PCI device. The timing requirement for system software generally allows for a margin of the order of tens of milliseconds as evident in several Linux subsystem drivers which use timeouts such as `msleep`.

However, realtime systems require timeliness in addition to correctness constraints. In particular, Realtime systems have the following properties:

- **Realtime systems have both functional and timing constraints:** Being 'realtime' means that if a set of tasks are schedule-able, then the tasks are schedulable, honouring the priority, deadlines and worst-case execution time conditions. Thus, both functional correctness and timeliness are equally key to a robust realtime system and one which fails to meet either of the two correctness constraints is said to have a failure.

Timing Constraints have two components - Predictability & Low Latency.[3]

For Predictability, the following conditions should be met:

- The time for every method should be predictable and should be constant irrespective of system load.
- Non-pre-emptible portions of methods should be short and deterministic.
- Interrupt handlers are scheduled and executed at appropriate priority.
- Interrupt latency is predictable and deterministic.

For Low Latency, the following conditions should be met:

- Software aims at reducing context-switch time, interrupt latency, and synchronization time.

- **Realtime software today requires specialized Oses for development:** Because of all the conditions mentioned above, writing realtime systems

today require low-level support from the kernel containing specialized features such as static priorities, fast and predictable interrupt latency and pre-emption in the OS kernel. Static priorities are key to realtime scheduling algorithms such as Earliest Deadline First(EDF) and Rate Monotonic Algorithm (RMA) as these compute schedulability based on pre-assigned priorities.

Conventional Unix did not support these and this motivated scientists to come up with specialized kernels for Real-time software.

1.3 Reason for choosing Golang for this work

Golang was selected because of a number of reasons concerning its runtime environment and language specifications such as:[6]

- **Golang has features of a realtime programming language:** Golang, with its rich runtime has many features of a realtime OS such as the following:
 - Golang runtime possesses its own timer infrastructure which allows for programming execution time and deadlines for the tasks, as required for realtime programs.
 - Golang runtime has support for light-weight threading by means of goroutines and userspace scheduling. This also allows fast and predictable context switching, a key requirement as far as realtime systems are concerned.
 - Golang runtime has a work stealing scheduler.[7]In a work stealing scheduler, the underutilized processors take the initiative to *steal* work from other processors. This is in contrast with a work sharing scheduler, which always tries to distribute work among the different processors. The advantage of a work stealing scheduler is minimal thread migration, leading to lower communication complexity and space.[4]

- **Support for most CPU-OS combinations:** The runtime environment of the Golang programming language has support for most of the OSes and CPU architectures in use today which could be used to converge realtime software development with generic software development.[6]
- As of 2017, Golang is an emerging modern programming language for programming cyber-physical platforms(CPS).[5]

1.3.1 Issues with Current Golang for Realtime Programming

While there are a number of positives for choosing Golang, the stock Golang cannot be directly used in a realtime setting, because of the lack of realtime considerations in the runtime scheduler.

- **No notion of static priorities:** Golang runtime currently does not possess programmability for the priority and all goroutines from the user program have the same priority. Static priorities are important so that schedulability tests can determine if the set of realtime tasks are permissible.
- **No notion of deadlines:** As seen in section 3.3.2, Golang has low latency of context-switching which is promising as far as timing is concerned. However, missing deadlines is synonymous with failure in realtime software and this idea is central to a realtime system.
- **No notion of execution time:** Realtime systems use the worst-case execution time to decide the next task to execute as it is a part of the schedulability test. Golang runtime presently does not have a facility to program execution times into the program. This is important for schedulability tests to take into consideration and also since predictability is central to a realtime system.

1.4 Problem statement

In this work, I will describe the approach to develop a software platform built on the runtime environment of Golang. This software platform is able to execute realtime tasks on a commodity OS like Linux to support the co-existence of realtime and non-realtime software components.

1.5 Contributions

This research makes two key contributions:

- A software runtime patch for Golang that orchestrates realtime task scheduling on a standard Linux-based system. The primary insight that drives my work is the run-time scheduler of Golang release 1.8, and can be used to schedule realtime as well as non-realtime tasks
- An evaluation of the concurrent garbage collector in Golang to understand the viability, implications and challenges of garbage collection in the realtime setting.

1.6 Layout of Thesis

The reminder of the thesis is organized as follows: Chapter 2: *Background* starts with the history of Golang and describes the key features which make it a promising systems programming language. This chapter also gives an overview of the entire software stack of Golang focussing on its powerful and rich runtime environment. Chapter 3: *The Golang Runtime Scheduler* explains in detail how goroutines are scheduled by the Golang runtime in coordination with the OS scheduler and shows how it can be tweaked to include realtime programs. Chapter 4: *The Golang Garbage Collector* explains how Golang's garbage collection works and studies the implication of garbage collection in a realtime environment. Chapter 5: *Conclusion* discusses the key contributions of this research, lessons learnt and the future work in this space.

Bibliography

- [1] "Cilk." Wikipedia. Wikimedia Foundation, 15 Feb. 2017. Web. 15 July 2017, <https://en.wikipedia.org/wiki/Cilk>
- [2] "Runtime System." Wikipedia. Wikimedia Foundation, 30 June 2017. Web. 18 July 2017, https://en.wikipedia.org/wiki/Runtime_system
- [3] "Notes from UC San Diego CSE237B, Spring 2016." Professor Rajesh Gupta, 15 April 2016. Print. 17 July 2017.
- [4] "Scheduling multithreaded computations by work stealing." Robert D. Blumofe and Charles E. Leiserson. J. ACM 46, 5 September 1999, 720-748, <http://dx.doi.org/10.1145/324133.324234>
- [5] Donovan, Alan A. A., and Brian W. Kernighan. The Go Programming Language. New York: Addison-Wesley, 2016. Print.
- [6] The Go Programming Language Specification. N.p., 18 Nov. 2016. Web. 15 Mar. 2017, <https://golang.org/ref/spec>
- [7] "Package Runtime." Runtime - The Go Programming Language. N.p., n.d. Web. 15 Apr. 2017, <https://golang.org/pkg/runtime/>

Chapter 2

Background

2.1 Introduction to Golang

Golang was first launched in the year 2009 by three Google engineers Robert Griesemer (known for the Java Hotspot Virtual Machine), Rob Pike (known for Plan 9 and Unix at Bell Labs), and Ken Thompson (known for Unix and C at Bell Labs).

Golang is an open-source language, distributed with the BSD license. In syntax and form, Golang bears a similarity to the C programming language. In addition, the syntax is made more intuitive, concise, and clean. For its concurrency primitive, Golang is influenced by Erlang, which was based on Tony Hoare's idea of communicating sequential processes (CSP).[10]

Since May 2010, Golang has been used in production at Google for back-end infrastructure, e.g. writing programs for administration of complex environments.

This chapter will give an overview of the language with an emphasis on the runtime environment that forms the cornerstone of my thesis.

2.2 Goals of Golang

There are two prevalent trends in software industry today. First, the number of cores in CPUs are on the rise. Second, high-performance software is written in a language such as C++, which is a very complex language with a detailed specification running to over a thousand pages.[9]

It is in such an environment that Golang has been developed by its designers. The core goal is to combine the efficiency, speed, and safety of a strongly-typed language (such as C++) with the ease of programming of a dynamic language (such as Python or Ruby). The second goal of Golang is to cater to the recent trend of distributed and multicore computing by providing support for networking, concurrency, and parallelism. The third goal is to decrease time taken to compile software, to combat long test-develop cycles endured by C++ developers at Google.

Furthermore, Golang also targets the execution speed of C/C++ when it comes to executing native code. While C++ has provides bare minimal memory management and expects the developer to manage it in his code, Golang manages memory in its runtime environment.

2.3 Characteristics of Golang

- Golang is an imperative (procedural, structural) language built with concurrency in mind. Golang is not object-oriented in the traditional sense, as it does not have the concept of classes or inheritance. In turn, Golang treats functions as first-order objects, supports interfaces and higher-order functions.
- Golang is statically typed, thus a safe language, and compiles to native code, which provides very high execution speeds. However, it also has *support for dynamic typing*, via automatically determining types for an interface (called static duck typing), which appeals to Python, Ruby, and Perl programmers.

- Golang has support for **cross-compilation**, which enables it to be compiled on a host of operating systems for a wide variety of processor architectures.
- Golang also exhibits some selected features of a **functional languages** such as closures and iterators. However, it omits some features of functional languages such as maps and generics.

2.4 Where is Golang used?

Being a systems programming language at heart, Golang is used in high-end distributed computing, such as web servers, storage area networks, and even for multi-player game development where massive parallelism and concurrency is common.

Golang is also used in complex event processing applications, which are used for data analytics in Internet-of-things applications. Furthermore, Golang can also be used for general software development, such as text processing, script writing for data analysis, and front-end development. A number of companies other than Google have also moved onto Golang for their back-end software development.[8]

2.5 Brief overview of Golang

In Golang, as in any other programming language, one builds large programs from small set of basic constructs. There are variables that store data values, and there are data types that are used depending on the specific data and program. Golang has operators, such as addition and subtraction, which are used to chain simple expressions into larger, more complex expressions. These expressions can be used in control flow statements (i.e. if) or in looping structures (i.e. for) to redirect the flow of the program. Several program statements are grouped into functions for isolation and reuse. Functions are combined into source files, and many such source files are present in a single package. Packages in Golang serve a purpose similar to that of libraries or modules in other

languages.[2]

Golang offers a variety of ways to organize the data, and its data types fall into four categories: basic types, aggregate types, reference types, and interface types. Basic types include numbers, strings, and booleans. Aggregate types combine basic types into a larger entity. Examples of aggregate types include arrays and structs. Reference types, as the name suggests, are defined such that when an operation is applied to one reference type, the change is applied to all the copies of that reference. Examples of reference types include pointers, slices, maps, and functions. Lastly, interface types express generalizations or abstractions about the behaviours of other data types. This adds to the flexibility of the language since functions written this way can be redefined and re-implemented.

Finally, concurrency is the expression of a program as a composition of several autonomous activities. Golang has inherent support for concurrency using its primitives of goroutines and channels.

Golang enables two styles of concurrent programming.[2; 3]

- Using goroutines and channels, which are inspired from Hoare's idea of communicating sequential processes (CSP), a model of concurrency in which values are passed between independent activities(goroutines). These form a part of the language specification.[10]
- Using mutexes and waitgroups, which are derived from the traditional method of shared memory multithreading. The support for these are part of the sync package.

2.6 Background in Golang

Golang has its a C-like syntax and has constructs such as channels and goroutines. The simplicity of Golang makes writing programs very easy compared to other systems programming languages. We will discuss some of the constructs with an example, and note the features of the language with it.

Hello World

Golang organizes the programs into packages, and has import statements at the top of the program. Also, Golang flags redundant imports as an error; for example, if we were to import the OS package in the hello world program, it would be an error. By restricting imports, the program becomes very efficient.

```
package main
import "fmt"
func main() {
    fmt.Printf("hello , world\n")
}
```

Listing 2.1: Hello World in Golang

There are two methods to run a go program such as the one shown in Listing 2.1 :

1) Using the go run command:

```
go run hello_world.go
```

2) Building and running an executable:

```
go build hello_world.go
./hello_world
```

Multiple Return Values

Golang offers support for multiple return values from a function. This can be used in scenarios, such as opening a file wherein we can return the error value and the file handle in one statement. An example is shown in Listing 2.2.

Goroutines

Goroutines are primitives in Golang that enable support for concurrency. By default, functions will not run as a goroutine, unless we invoke it specifi-

cally using the `go` keyword, such as `go f()`. The benefit of using a goroutine is that it executes concurrently with the calling functions as is shown in Listing 2.3.

```
package main
import (
    "fmt"
    "os"
)
func main(){

    fmt.Println("Going to open a file ...")
    f, err := os.Create("sample.txt")
    if err != nil {
        fmt.Println("Could not create. Going to
            panic")
        panic(err)
    }

    byte_array := []byte{1,2,3,4,5}
    g, err := f.Write(byte_array)

    if err != nil {
        fmt.Println("Could not write into the file
            . Going to panic")
        panic(err)
    }

    fmt.Printf("Wrote %d bytes into the file\n", g)
}
```

Listing 2.2: Using An Example showing the use of Multiple Return Values in Golang

```
package main

import (
    "fmt"
    "time"
    "math/rand"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println("Goroutine:", n, "iteration:",
            i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i) // run it in the background
        //f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

Listing 2.3: An Example showing asynchronous nature of goroutines and the concurrency model

When we run `f` as a goroutine, the runtime environment interleaves the execution of the different goroutines. The execution order of the various goroutines is decided by the runtime scheduler. In the case shown in Fig 2.3, the goroutines do not block and run asynchronously.

Channels

Channels allow two goroutines to communicate with one another and synchronize their execution to achieve *Synchronization by communication*. In the program shown in Listing 2.4, we have two sender goroutines (ping and pong) and one receiver (goroutine print). For a successful execution, the print must be present to receive when ping tries to send a message. The ping waits (is blocked) until print has successfully received the message, before pong attempts to send the next message.

Channels can be directional

The direction for a channel can be specified when it is used as a function parameter. This increases type-safety of the program. The program in Listing 2.5 shows an example of passing a message through a set of two channels.[7]

```
package main
import (
    "fmt"
    "time"
    "math/rand"
    "strings"
    "strconv"
)
func ping(c chan string){
    for i:=0; i < 10; i++ {
        istr := strconv.Itoa(i)
        str := []string{"ping", istr}
        c <- strings.Join(str, ":")
    }
}
```

```
func pong(c chan string){
    for i:=0; i < 10; i++ {
        istr := strconv.Itoa(i)
        str := []string{"pong", istr}
        c <- strings.Join(str, ":")
    }
}

func printr(c chan string){
    for {
        message := <- c
        fmt.Println(message)
        time.Sleep(time.Millisecond * time.
            Duration(rand.Intn(250)))
    }
}

func main(){
    var c chan string = make(chan string)

    go ping(c)
    go pong(c)
    go printr(c)

    var input string
    fmt.Scanln(&input)
}
```

Listing 2.4: An Example showing the use of channels to achieve synchronization in golang's concurrency model.

Channels Plus Goroutines: Select

`select` is a very unique feature of Golang. In the example in Listing 2.6, we receive a value from each of the two channels and `select` is used to receive them simultaneously as it arrives. Once the values arrive, we print them. Note that the execution time is same as the execution time of the longest duration channel operation, being 2 seconds for this example.

```

/* The Channel Arrangement is as follows:
                                     _____
"passed message" ———> channel 1 ———> channel 2
                                     _____

*/
package main
import "fmt"
func ping(receiveonly chan<- string , message string){
    receiveonly <- message
}
func pong(sendonly <-chan string , receiveonly chan<-
string){
    mesg := <-sendonly
    receiveonly <- mesg
}
func main(){
    channel1 := make(chan string ,1)
    channel2 := make(chan string ,1)
    ping(channel1 , "passed message")
    pong(channel1 , channel2)
    fmt.Println(<-channel2)
}

```

Listing 2.5: An Example showing the use of directionality in channels. Note that the channels used in the example are buffered. This is done to enable a store-and-forward mechanism.

Channels Plus Goroutines: Select

`select` is a very unique feature of Golang. In the example in Listing 2.6, we receive a value from each of the two channels and `select` is used to receive them simultaneously as it arrives. Once the values arrive, we print them. Note that the execution time is same as the execution time of the longest duration channel operation, being 2 seconds for this example.

```
/* select is used to wait and receive from both
channels simultaneously
*/

package main
import (
    "time"
    "fmt"
)
func main(){
    channel1 := make(chan string)
    channel2 := make(chan string)
    /*Anonymous Functions*/
    go func(message string){
        time.Sleep(time.Second * 1)
        channel1 <- message
    }("first message")
    go func(message string){
        time.Sleep(time.Second * 2)
        channel2 <- message
    }("second message")
    for i := 0; i < 2; i++ {
        select {
            case msg1 := <-channel1:
```

```

        fmt.Println("Received", msg1)
    case msg2 := <-channel2:
        fmt.Println("Received", msg2)
    }
}
}
}

```

Listing 2.6: An Example showing the use of select statement

Another scenario where select is useful is to implement timeouts in Golang. Inside a select statement, we can use the `time.After(timeout)` to kickstart a timer which will send values to a channel after the timeout has elapsed. If the operation we care about completes before this timeout elapses, we say that the operation has succeeded. If on the other hand, the operation takes longer than the duration specified in the `time.After` call, the operation has timed out and we take appropriate action. An example of implementing timeouts using select statement is shown in Listing 2.7. [7]

```

/*select is used to timeout here
The first function call takes 2 seconds, which
timesout since the select picks the first
value that is ready.

The second function call takes 2 seconds, which
is printed since the timeout threshold is
3 seconds
*/

package main

import (
    "time"

```

```

        "fmt"
    )

func main(){
    channel1 := make(chan string , 1)
    go func(){
        time.Sleep(time.Second * 2)
        channel1 <- "First Ping"
    }()
    select{
    case result := <-channel1:
        fmt.Println(result)

    case <-time.After(time.Second * 1):
        fmt.Println("First Ping: Timed out")
    }

    channel2 := make(chan string , 1)
    go func(){
        time.Sleep(time.Second * 2)
        channel2 <- "Second Ping"
    }()
    select{
    case result := <-channel2:
        fmt.Println(result)
    case <-time.After(time.Second * 3):
        fmt.Println("Second Ping: Timed out")
    }
}

```

Listing 2.7: An Example showing the use of select statement to implement timeouts

2.7 Golang runtime environment

Similar to C's glibc and Java's Java Virtual Machine(JVM), Golang has a library known as the runtime library, which supports in running the program. The runtime library is located at `src/runtime` in the source code of Golang, and contains files managing a plethora of tasks for Golang programs, such as scheduling, stack management, garbage collection and interaction with the OS via system calls. The architecture of the runtime environment is shown in Fig 2.1.[4; 5]

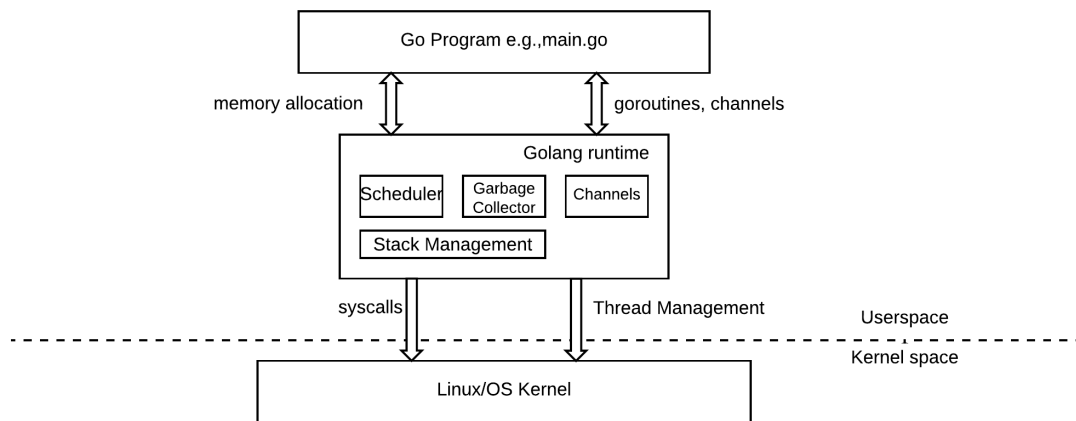


Figure 2.1: Block diagram of the runtime environment and its interactions with the OS

In this thesis, we will study the runtime environment of Golang, with a focus on the runtime scheduler, and leverage it to attain the desired realtime properties.

2.8 Benchmarking Timing of Golang Applications

Benchmarking and Profiling are two methodologies used to measure the performance of a specific program. While the two terms provide similar information, the methodologies are distinctive. Simply put, benchmarking provides

the relative performance of a program in relation to a baseline, whereas profiling is the objective and absolute measurement of the program performance.

Benchmarking is the practice of stress-measuring the performance of a software, wherein the stress test is static and deterministic.[8] For example, an OpenGL benchmark on a smartphone, such as Taiji[®], can be used to measure the 3D graphics performance of the Android OS on any embedded device. In Golang, a benchmark is a special function that is prefixed with the word `Benchmark` and takes `b *testing.B` as arguments. The benchmark functions are typically defined in a separate file from the program that they are evaluating. For example, the Golang runtime source code has a separate file named `stack_test.go` that contains the benchmark and test functions to benchmark the file `stack.go`.

In contrast, profiling is measuring the absolute speed and efficiency and is commonly used to find out critical regions in code which could mean that they consume more CPU cycles or use too much memory.[2] Profiling commonly involves sampling a number of events by sending the signal `SIGPROF` to the program being profiled. Signals, we recall, are used by the OS to communicate with the application process. In our specific case of Golang runtime, the signal is sent by the userspace to the runtime and the runtime has a function handler `func sigprof(pc, sp, lr uintptr, gp *g, mp *m)` defined to handle this at `proc.go` on line 3180.

As Golang researcher Dave Cheney covered in his talk at the Golang UK Conference 2016[8], there are 7 ways of profiling Golang programs:

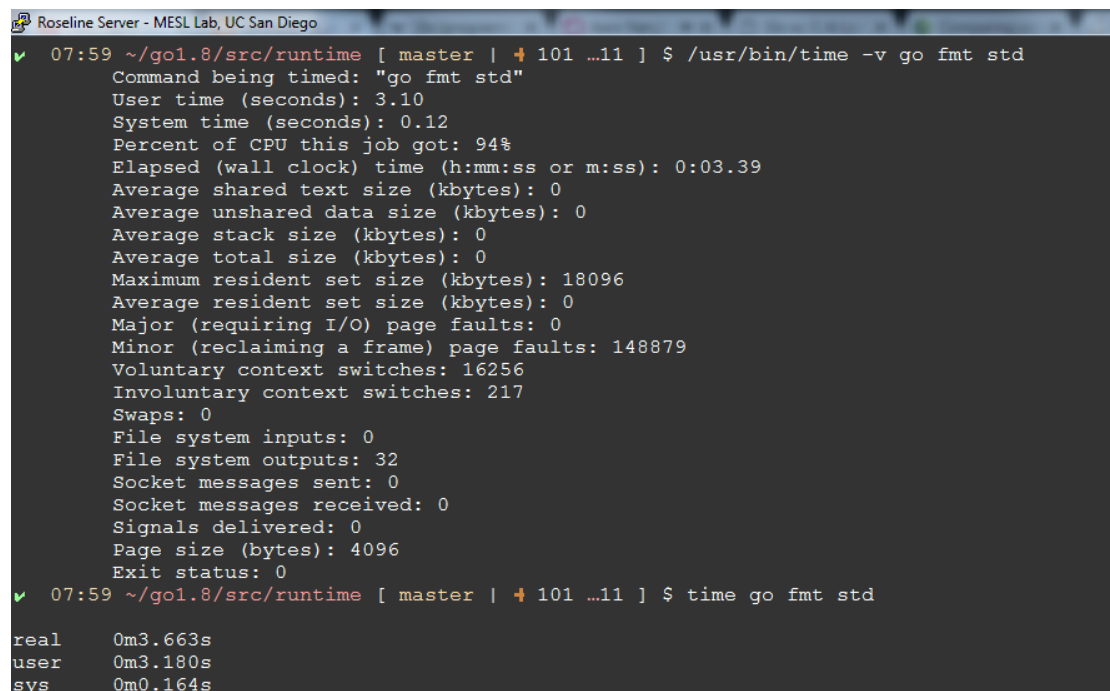
1. GNU/Linux `time` command
2. Runtime Environment variable `GODEBUG`
3. Runtime Environment tool `runtime/pprof`
4. Tracing Facility `go tool trace`
5. For debugging server applications, `debug/pprof`
6. GNU/Linux `Perf`
7. `Flamegraph`

We used the first four techniques for analyzing the scheduler but found `runtime/`

pprof and go tool trace to be the most useful as it invokes the execution tracer in the runtime environment to give detailed traces at a very high resolution.

Time Command

We can use the GNU/Linux command `/usr/bin/time -v <command>` to give us (1) the timing details of the system at a high level, classified as real, user, and system times; and (2) other statistics, such as context switches and page faults. There is also the less powerful shell built-in command, which gives us only the time spent in the system, user, and real modes. The GNU/Linux manual states that the precision of this command is unspecified, but is sufficient to express the clock tick accuracy. In Fig 2.2, we have an example execution of the time command for go fmt operation. go fmt std is a command to perform recursive directory traversal, and to format the *.go files in accordance with Golang stylistic guidelines.[1]



```

Roseline Server - MESL Lab, UC San Diego
07:59 ~/go1.8/src/runtime [ master | + 101 ...11 ] $ /usr/bin/time -v go fmt std
Command being timed: "go fmt std"
User time (seconds): 3.10
System time (seconds): 0.12
Percent of CPU this job got: 94%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:03.39
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 18096
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 148879
Voluntary context switches: 16256
Involuntary context switches: 217
Swaps: 0
File system inputs: 0
File system outputs: 32
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
07:59 ~/go1.8/src/runtime [ master | + 101 ...11 ] $ time go fmt std
real    0m3.663s
user    0m3.180s
sys     0m0.164s

```

Figure 2.2: Example execution of the time command for go fmt operation.

We note that the time command gives us a precision of a 1 millisecond for measuring time. time gets the timing information from underlying GNU/Linux

timeval structure and performs truncation and/or rounding but modifying precision of time requires recompiling the kernel infrastructure. Thus, we can use this tool to obtain an overall diagnosis of the program timing, but to obtain fine-grained information of timing, we need richer tools, as will be described in the following sections.

GODEBUG

GODEBUG is an environment variable, which is used to capture the user code interactions with the runtime environment such as goroutine scheduling, garbage collection, and memory allocation/deallocation. The package runtime documentation describes flags that can be used in conjunction with the GODEBUG environment variable.[6] As an example, we have a test code containing two functions. One is the main function, which spawns ten goroutines, each of which counts to 10 billion in the second function, and calls done to signal its completion back to main function. The scheduler traces are displayed in Fig 2.3.

Using GODEBUG, we can study status of the goroutines, worker threads, and contexts in terms of their state transitions, as well as their interactions with the garbage collector. Here, we collect the scheduler traces every 1000ms from the runtime. scheddetail is a switch used to emit detailed multiline information every 1000ms. The most precision we can get of timing from the scheduler runtime variable GODEBUG is 1 millisecond, which is identical to the tools we have seen so far.

pprof

pprof is a tool derived from the Google Performance Tools suite, and is built into the Golang runtime.[4] pprof works in tandem with go test tool. As noted in the documentation, go test tool has built-in support for three styles of profiling:

1. **CPU Profiling:** This is used to find the critical code that consumes the most

```

Roseline Server - MIESL Lab, UC San Diego
16:27 ~/BO/spc/schedexpt $ env GODEBUG=schedtrace=1000,schedetail=1,./schedexpt
SCHED 0ms: gomaxprocs=4 idleprocs=2 threads=2 spinningthreads=1 idletheads=1 runqueue=0 gwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
P0: status=1 schedtick=1 syscalltick=0 m=0 runsize=0 gfreecnt=0
P1: status=1 schedtick=0 syscalltick=0 m=2 runsize=0 gfreecnt=0
P2: status=0 schedtick=0 syscalltick=0 m=-1 runsize=0 gfreecnt=0
P3: status=0 schedtick=0 syscalltick=0 m=-1 runsize=0 gfreecnt=0
M3: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true locked=-1
M2: p=1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M1: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M0: p=0 curg=14 mallocing=0 throwing=0 preemptoff= locks=2 dying=0 helpgc=0 spinning=false blocked=false locked=-1
G1: status=4(semacquire) m=-1 lockedm=-1
G2: status=1() m=-1 lockedm=-1
G3: status=4(GC sweep wait) m=-1 lockedm=-1
G4: status=1() m=-1 lockedm=-1
G5: status=1() m=-1 lockedm=-1
G6: status=1() m=-1 lockedm=-1
G7: status=1() m=-1 lockedm=-1
G8: status=1() m=-1 lockedm=-1
G9: status=1() m=-1 lockedm=-1
G10: status=1() m=-1 lockedm=-1
G11: status=1() m=-1 lockedm=-1
G12: status=1() m=-1 lockedm=-1
G13: status=4(sleep) m=-1 lockedm=-1
G14: status=3() m=0 lockedm=-1
SCHED 1008ms: gomaxprocs=4 idleprocs=0 threads=6 spinningthreads=0 idletheads=1 runqueue=0 gwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
P0: status=1 schedtick=1 syscalltick=2 m=0 runsize=2 gfreecnt=0
P1: status=1 schedtick=10 syscalltick=0 m=3 runsize=0 gfreecnt=0
P2: status=1 schedtick=2 syscalltick=0 m=5 runsize=0 gfreecnt=0
P3: status=1 schedtick=1 syscalltick=0 m=4 runsize=4 gfreecnt=0
M5: p=2 curg=9 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M4: p=3 curg=4 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M3: p=1 curg=10 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M2: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=true locked=-1
M1: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 helpgc=0 spinning=false blocked=false locked=-1
M0: p=0 curg=5 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 helpgc=0 spinning=false blocked=false locked=-1
G1: status=4(semacquire) m=-1 lockedm=-1
G2: status=4(force gc (idle)) m=-1 lockedm=-1
G3: status=4(GC sweep wait) m=-1 lockedm=-1
G4: status=2(sleep) m=4 lockedm=-1
G5: status=2(sleep) m=0 lockedm=-1
G6: status=1(sleep) m=-1 lockedm=-1
G7: status=1(sleep) m=-1 lockedm=-1
G8: status=1(sleep) m=-1 lockedm=-1
G9: status=2(sleep) m=5 lockedm=-1
G10: status=2(sleep) m=3 lockedm=-1
G11: status=1(sleep) m=-1 lockedm=-1
G12: status=1(sleep) m=-1 lockedm=-1
G13: status=1(sleep) m=-1 lockedm=-1
G14: status=4(timer goroutine (idle)) m=-1 lockedm=-1

```

Figure 2.3: Example execution of GODEBUG command to capture scheduler traces along with the timestamps.

CPU time. It is captured using a command similar to `go test -cpuprofile = prof.cpu`, which essentially compiles all the `*_test.go` files in the current directory, and writes the CPU profile to the file `cpu.out`.

2. **Memory Profiling:** This is used to find the critical code that consumes the most memory. Like CPU profiling, memory profiling is captured using the flexible command `go test -memprofile=prof.mem`.

3. **Block Profiling:** Block profiling is a very unique form of profiling, and is used to find and debug situations, such as a goroutine waiting for a long time on a value from a channel or any synchronization primitive (i.e. mutexes or semaphores). Like CPU and memory profiling, timing profiling is captured using the command `go test -blockprofile=prof.block`. Block profiling can be very useful for determining concurrency bottlenecks in the application, because it can show us when a large number of goroutines were blocked when they should have made progress.[8]

The files `cpu.out`, `mem.out`, and `block.out` can be used by a tool, such as `pprof`, to give additional insight on the profile. `pprof` is commonly referred to in the Golang research community as the Swiss Army Knife of all the performance tools. The complete commands used are as shown in Fig 2.4. The benchmark chosen is in `src/runtime/proc_test.go` and the `go test` command is run inside the `src/runtime` directory to create a file `runtime.test`. `runtime.test` and the profile file `prof.cpu` is used by the `pprof` tool to interpret the data.

Example of a Benchmark: Parallel Goroutine creation

Parallel Goroutine is similar to the `pthread_create()` and the `pthread_join()` in C++. We spawn `GOMAXPROCS` number of goroutines, each of which communicates its completion by writing `true` to a boolean channel, which is unbuffered. In other words, the sending goroutine is held blocked until the receiving goroutine is present, and vice versa. The `pprof` tool can also be used to visualize the runtime callgraph, as shown in Fig 2.5. From the callgraph, one can quickly identify the goroutine that is consuming the most CPU time, based on the box size and time stamps.

```

Roseline Server - MESL Lab, UC San Diego
15:12 ~/go1.8/src/runtime [ master | ↑ 188 ...14 ] $ go test -run=$ -bench=BenchmarkCreateGoroutinesParallel -cpuprofile=prof.cpu
BenchmarkCreateGoroutinesParallel-4 30000000
PASS
ok runtime 1.240s
15:12 ~/go1.8/src/runtime [ master | ↑ 188 ...14 ] $ go tool pprof runtime.test prof.cpu
Entering interactive mode (type "help" for commands)
(pprof) top
2070ms of 2590ms total (79.92%)
Dropped 3 nodes (cum <= 12.95ms)
Showing top 10 nodes out of 32 (cum >= 1200ms)
flat flat% sum% cum cum%
540ms 20.85% 20.85% runtime.caagstatus
420ms 16.22% 37.07% runtime.newprocl
240ms 9.27% 46.33% runtime.(*guintptr).cas
200ms 7.72% 54.05% runtime.goexit0
160ms 6.18% 60.23% runtime.schedule
150ms 5.79% 66.02% runtime.execute
110ms 4.25% 70.27% runtime.gfget
100ms 3.86% 74.13% runtime.systemstack
80ms 3.09% 77.22% runtime.mcall
70ms 2.70% 79.92% runtime.newproc
(pprof)

```

Figure 2.4: Example execution of `go test` and `pprof` tools to find functions consuming most time of the CPU. We can see that the function `casgstatus()` and `newprocl()` are consuming most CPU time here which is expected as the former is called when the status of a goroutine changes and the latter is called whenever a goroutine is created. These functions are defined in the runtime environment in the file `proc.go`.

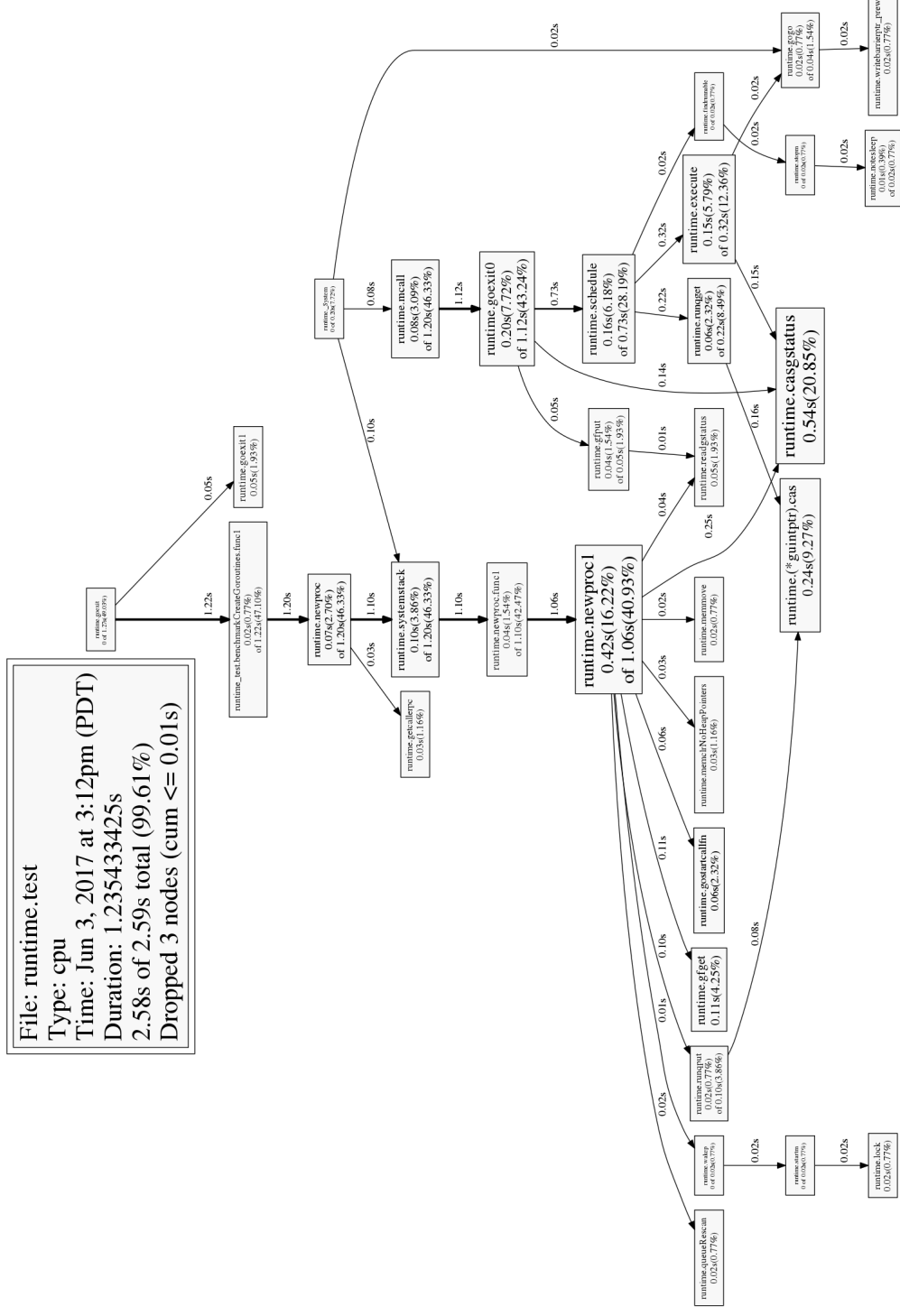


Figure 2.5: Example execution of png command to find functions consuming most time of the CPU as well as an overview of the callgraph of the codebase. We can see that the function `casgstatus()` and `newproc1()` are consuming most CPU time here as well which is shown by the big size of the boxes representing these two functions.

Note that the Parallel Goroutine creation is an example of a benchmark. A benchmark is a special type of function that is placed in a file ending with `*_test.go`, and it is run using the `go test` tool. Benchmarks and test function are run using the `testing/` framework located in `src/testing` directory of the Golang source, and do not require a `main()` function.

On the other hand, to profile user programs with a `main()`, we can use the `go tool trace` command. Lastly, we note that the `time` command gives us a precision of a 1 nanosecond for the timing information. This high resolution makes `pprof` a valuable tool for debugging issues related to performance and timing.

go tool trace

`go tool trace` provides a lot of visual information to the user, but is not very well documented apart from a couple of talks by Dimitry Vyukov and Rhys Hiltner. [11] We can start capturing the trace using `runtime/trace` package by adding `trace.Start(<file>)` and `trace.Stop(<file>)` statements around the points of the code we want to analyze. Note that this requires importing the `os` package as well since we need to write the traces to a file. An example code snippet is shown in Fig 2.13 and the output is shown in Fig 2.6.

```
package main

import (
    "os"
    "runtime/trace"
    "sync"
    "time"
)

func main() {
    //trace.out is the file that will contain our

```

```
    trace
f, err := os.Create("trace.out")

//Error handling in case the returned file
    scriptor returned is invalid
if err != nil{
    panic(err)
}
defer f.Close()

//Start the tracing facility
err = trace.Start(f)
if err != nil{
    panic(err)
}

//Stop the tracing when main exits
defer trace.Stop()

//WaitGroup wg waits for goroutines to finish
var wg sync.WaitGroup
wg.Add(10)

for i := 0; i < 10; i++ {
    go work(&wg)
}

//wait for Done from each of the goroutines.
wg.Wait()

// Sleep so that the flow does fall out of main()
```

```

        before the goroutines return
        time.Sleep(3 * time.Second)
    }

func work(wg *sync.WaitGroup) {
    time.Sleep(time.Second)

    var counter int
    for i := 0; i < 1e10; i++ {
        counter++
    }

    wg.Done()
}

```

Listing 2.8: Using the `go tool trace` facility to log the tracing output to `trace.out`. We can then view the output using `go tool trace trace.out`

The tools `perf`, `flamegraph` and `debug/pprof` can also be used to obtain similar type of insight into the runtime environment.

2.9 Conclusion

In this chapter, we saw an overview of the Golang programming language and studied the different features of the language. Furthermore, we also saw some tools which can be used for performance evaluation in Golang which is key in timing measurement. This sets the stage for exploring the runtime environment beginning with the scheduler in the next chapter.

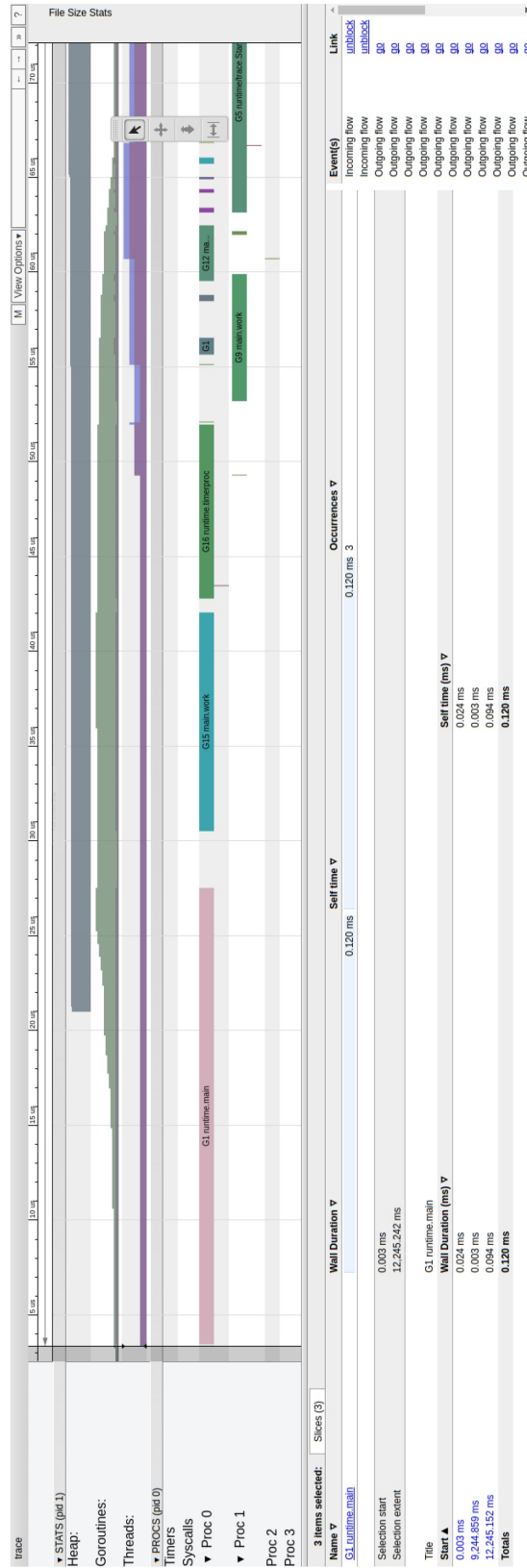


Figure 2.6: go tool trace output showing the timeline of execution of the goroutines. We see that the 10 spawned goroutines are indicated in the right bottom of the graph.[7]

Bibliography

- [1] Go Programming Language Specification. N.p., Nov. 2016. Web. Mar. 2017, <https://golang.org/ref/spec>
- [2] Donovan, Alan A. A., and Brian W. Kernighan. The Go Programming Language. New York: Addison-Wesley, 2016. Print.
- [3] "Concurrency Is Not Parallelism." The Go Blog. N.p., 16 Jan. 2013. Web. 15 Apr. 2017, <https://blog.golang.org/concurrency-is-not-parallelism>
- [4] "Package Runtime." Runtime - The Go Programming Language. N.p., n.d. Web. 15 Apr. 2017, <https://golang.org/pkg/runtime/>
- [5] Deshpande, Neil, Erica Sponsler, and Nathaniel Weiss. "Analysis of the Go runtime scheduler." (2011): n. pag. 12 Dec. 2011. Web. 15 Apr. 2017.
- [6] McGranaghan, Mark. "Go by Example." Go by Example. Mark McGranaghan, 20 June 2017. Web. 15 July 2017, <https://gobyexample.com/>
- [7] Golang UK Conference 2016 - Dave Cheney - Seven Ways to Profile Go Applications. Dir. Dave Cheney. YouTube. N.p., Sept. 2016. Web. May 2017, https://www.youtube.com/watch?v=2h_NBFrciI
- [8] "Golang/go." GitHub. Golang Wiki, 1 Jan. 2017. Web. 18 July 2017, <https://github.com/golang/go/wiki/GoUsers>
- [9] "The Standard." Standard C++. N.p., n.d. Web. July 2017, <https://isocpp.org/std/the-standard>
- [10] "Frequently Asked Questions (FAQ)." Frequently Asked Questions (FAQ) - The Go Programming Language. N.p., n.d. Web. July 2017, <https://golang.org/doc/faq>

Chapter 3

The Golang Runtime Scheduler

Process is the abstraction used by the Operating System(OS) for managing tasks; in turn, process scheduling is the methodology used by the OS to compute and complete those tasks through execution. The tasks may be thought of as the smallest executable unit of a user program. Process scheduling is about partitioning the finite resource of time that a CPU has into executing different tasks, so that each of the the task meets its goals.

The concept behind a scheduler is simple. If there are tasks that are *ready to run*, and there is an idle CPU, then the CPU should be executing the task. Conversely, if there are more *runnable* tasks than CPUs in a machine, then some of the tasks will be *waiting to run*. The primary job of a scheduler is to select the next task to be executed among a set of runnable tasks.

The first section of the chapter deals with the concepts used in scheduling, and how scheduling is carried out in the Golang runtime environment. The section is followed by an evaluation of the process scheduler used in Golang v1.8 by running benchmarks using the test utility. We then define realtime scheduling, and explain how Golang's runtime environment may be tweaked for providing realtime scheduling. We call this new runtime environment *Realtime Golang*. We conclude the chapter with the details of Realtime Golang, a runtime system aimed at realtime task management.

Note that this chapter will be interchangeably using the terms 'process' or 'task'. If at any point of time, the distinction becomes important, it will be

called out. Furthermore, we will be using the terms ‘CPU’ and ‘processor’ to refer to a blackbox that can execute instructions in any user program.

3.1 Introduction to Scheduling

Scheduling is an interdisciplinary science drawing inputs from Mathematics, Computer Science and Engineering. The discipline of queuing theory and schedule-ability helps model the tasks in a system, and quantitatively analyze the effect of executing different tasks at different times. We will go over some design decisions and scenarios affecting scheduling as it will lead us to the changes that are required for making Golang realtime.

3.1.1 Scheduling Terminology

Multitasking is defined in the Oxford Dictionary as *to execute more than one program or task simultaneously*. On a computing level, a CPU is a resource, which can execute one instruction at a time. However, in an OS that supports multitasking, the CPU can execute a task for a finite period of time (called a *timeslice*), save its progress (called *context*) into the stack, and *context-switch* to another task. This process of interleaving multiple task executions onto a timeline gives the user the *illusion* of simultaneous execution of multiple tasks (called *concurrent tasks*).

Parallelism is another term that is interchangeably used with concurrency, although both are strictly different. Parallelism is *actually* executing multiple tasks at an instance of time. In computing terms, parallelism can be achieved with multiple CPUs. On the other hand, concurrency is more about the programming model to give the *illusion* of multitasking.

On a Uni-processor (single CPU) machine, we can have concurrency but no parallelism. We can also say that parallelism degenerates to concurrency on a Uni-processor machine. On the other hand, on a Multi-processor (multiple CPUs) machine, we can have concurrency and parallelism, which can be differentiated by the scheduler.[5]

Tasks may be classified as either I/O-bound or CPU-bound. I/O-bound tasks spend much time submitting and waiting on I/O requests, such as disk I/O or network I/O. Therefore, such tasks are runnable for only short durations. Examples of I/O-bound tasks include graphical user interface (GUI) applications that often wait on user interaction (even if a user is typing fast or moving the mouse as quick as he can, it is still an eternity for the processor in terms of scheduling). Conversely, CPU-bound tasks spend much of their timeslice performing computations using the CPU. An example of CPU-bound task is computing the prime factors of a large number.

Note that the given definitions of tasks are not mutually exclusive. A composite program may have CPU-bound parts, as well as I/O-bound parts. For example, the game, Angry Birds®[®], is I/O-bound when waiting for the user's input, and is CPU-bound when it performs pixel-by-pixel calculation of the bird's next location.

Thus, the scheduling policy in a system must attempt to satisfy two conflicting goals: *fast process response time (low latency) and maximal system utilization (high throughput)*.^[9] While providing the highest throughput and the lowest latency simultaneously is impossible, it is possible to engineer a trade-off among the two, and skew the OS towards favouring one over the other. The default scheduler in the Linux OS, known as the 'Completely Fair Scheduler' favours I/O-bound tasks over CPU-bound tasks, as the OS is tailored to usage in a desktop environment.

One way the Linux OS overcomes the conflicting tradeoff is by achieving modularity through the concept of **scheduler classes**. Scheduler classes enable various pluggable algorithms to coexist, with each algorithm scheduling its own type of processes. Each scheduler class has a priority. The base scheduler code, which is defined in `kernel/sched.c`, iterates over each scheduler class in order of priority. The highest priority scheduler class that has a runnable process wins, and selects who runs next.

In the Linux 4.10.14 mainline kernel, we have six scheduling classes: `SCHED_RR`, `SCHED_NORMAL`, `SCHED_FIFO`, `SCHED_BATCH`, `SCHED_`

IDLE, and SCHED_DEADLINE. Each thread in Linux has a scheduling policy and a static scheduling priority. The classes are defined at include/uapi/linux/sched.h and are defined their man pages as follows:

- SCHED_FIFO: First in-first out scheduling policy without any timeslices. When a SCHED_FIFO thread becomes runnable, it will be inserted at the end of the list for its priority.
- SCHED_RR: Round-robin scheduling which is a special case of SCHED_FIFO with the added notion of timeslices. When a SCHED_RR thread runs for a duration equal to its timeslice, it is put at the end of the list for its priority.
- SCHED_NORMAL: Default Linux time-sharing scheduling which is intended for all general-purpose threads in the OS. These threads use the notion of dynamic priority which is based on the nice values.
- SCHED_BATCH: Batch scheduling policy is similar to SCHED_NORMAL except that the scheduler assumes threads are always CPU-bound. This changes the wakeup behaviour of threads but is commonly used in non-interactive workloads.
- SCHED_IDLE: Very low priority job scheduling policy which is used only at static priority. Nice values have no meaning when this scheduling policy is used.
- SCHED_DEADLINE: Sporadic task model deadline scheduling which is based on the global earliest deadline first(GEDF) algorithm and constant bandwidth server(CBS). This makes assumptions that the threads do not communicate with one another and uses an admissibility test to determine schedulability. The parameters that this scheduler uses can be set using system calls. SCHED_DEADLINE is the class having highest priority among all the classes.

Table 3.1: Process State Codes used in standard Linux System

D	uninterruptible sleep (usually IO).
R	running or runnable (on run queue).
S	interruptible sleep (waiting for an event to complete).
T	stopped by job control signal.
t	stopped by debugger during the tracing.
W	paging (not valid since the 2.6.xx kernel).
X	dead (should never be seen).
Z	defunct ("zombie") process, terminated but not reaped by its parent.

3.1.2 States of a Task

A task on its way to submission to the scheduler can be in one of many states. On a running Linux system, the command `ps -el` is used to view the different tasks running on the system as shown in Figure 3.1. The second column refers to the state of the particular task, which can be any of the possible states enumerated in Table 3.1.

```

Roseline Server - MESL Lab, UC San Diego
✓ 16:22 ~/go1.8/src/runtime [ master | ..9 ] $ ps -el
F S  UID  PID  PPID  C  PRI  NI ADDR  SZ  WCHAN  TTY          TIME CMD
4 S  0    1    0  0  80  0 - 29965 -    ?      ?      00:00:20 systemd
1 S  0    2    0  0  80  0 -    0 -    ?      ?      00:00:00 kthreadd
1 S  0    3    2  0  80  0 -    0 -    ?      ?      00:00:01 ksoftirqd/0
1 S  0    5    2  0  60 -20 -    0 -    ?      ?      00:00:00 kworker/0:0H
1 S  0    7    2  0  80  0 -    0 -    ?      ?      00:16:26 rcu_sched
1 S  0    8    2  0  80  0 -    0 -    ?      ?      00:00:00 rcu_bh
1 S  0    9    2  0 -40 - -    0 -    ?      ?      00:00:00 migration/0
5 S  0   10    2  0 -40 - -    0 -    ?      ?      00:00:04 watchdog/0
5 S  0   11    2  0 -40 - -    0 -    ?      ?      00:00:04 watchdog/1
1 S  0   12    2  0 -40 - -    0 -    ?      ?      00:00:00 migration/1
1 S  0   13    2  0  80  0 -    0 -    ?      ?      00:00:15 ksoftirqd/1
1 S  0   15    2  0  60 -20 -    0 -    ?      ?      00:00:00 kworker/1:0H
5 S  0   16    2  0 -40 - -    0 -    ?      ?      00:00:04 watchdog/2
1 S  0   17    2  0 -40 - -    0 -    ?      ?      00:00:00 migration/2
1 S  0   18    2  0  80  0 -    0 -    ?      ?      00:00:01 ksoftirqd/2
1 S  0   20    2  0  60 -20 -    0 -    ?      ?      00:00:00 kworker/2:0H
5 S  0   21    2  0 -40 - -    0 -    ?      ?      00:00:04 watchdog/3

```

Figure 3.1: Example Screenshot of tasks in an Operating System.

3.1.3 Scheduling Types

Broadly speaking, OSes rely on scheduling to enable multitasking. Depending on how scheduling is implemented, Multitasking OSes are classified into two types: *Cooperative Multitasking* and *Preemptive Multitasking*. Cooperative Multitasking, as the term suggests, relies on two tasks to cooperate to give one another a timeslice to the processor. This process is called *yielding*, and is completely under the control of the task. The OS plays no role in determining the duration of a task timeslice; the downside of this approach is that poorly programmed or malicious tasks can monopolize the system. Preemptive Multitasking, on the contrary, relies on the OS to decide how long a particular task can use the CPU. In particular, it is the process scheduler that decides the timeslice allotted to different tasks. This process is often predetermined, and enables the OS to make global scheduling decisions more effectively. Most of the modern OSes, like Linux and Windows use Preemptive Multitasking, the only exceptions being Mac OS 9 (and earlier) and Windows 3.1 (and earlier).[9]

Scheduling can also be classified based on the timing requirements of tasks. Tasks that do not have stringent timing characteristics are grouped into the `SCHED_NORMAL` scheduler class, and the corresponding schedulers are termed *non-real time schedulers*. Tasks of realtime schedulers, on the other hand, are grouped into three scheduling classes - `SCHED_DEADLINE`, `SCHED_RR`, and `SCHED_FIFO`. Realtime scheduling policies in Linux provide *soft realtime* behaviour, wherein the OS attempts to schedule applications within timing deadlines, but the kernel does not guarantee success. The other type of realtime scheduling behaviour is called *hard realtime*, wherein the OS makes guarantees to schedule applications within the designated timing requirements. The Realtime scheduling policies in Linux provide only 'soft realtime' behaviour.

In this thesis, we will analyze the Golang runtime scheduler, and use it to simulate realtime behaviour from the runtime environment.

3.2 Scheduling in the Golang runtime environment

The Golang runtime environment has its own scheduler to distribute ready-to-run goroutines over worker threads. The scheduler is based on a randomized work-stealing algorithm and is *partially pre-emptive*.^[13]

3.2.1 Why does scheduling have to be done at the user-level runtime environment?

Before we look at the scheduling strategy employed in the Golang runtime environment, we need to understand why we need scheduling in the first place. We need to justify the need for a scheduler in the userspace when there is already a scheduler in the OS kernel, which is an ever-increasingly mature piece of software. In this regard, let us have a look at the relative merits and demerits of user-level and kernel-level threads.

When the system uses **kernel-level threads**, the kernel knows about and manages the threads, and we no longer need a runtime system. The kernel has a thread table that keeps track of all the threads in the system in addition to the existing process table. This leads to the following merits and demerits:

Merits of Kernel-level threading

1. As the kernel knows everything about the threads, the kernel may decide to give more time to a process having the higher number of threads.
2. For applications that block, kernel-level threads are favourable. This is because the kernel is aware of threads that are idle threads or I/O bound threads. This way it can make an intelligent decision to schedule them. On the other hand, user-level threads are not visible to the kernel and the kernel can end up scheduling a user-level thread that is waiting on a disk or network I/O.

Demerits of Kernel-level threading

1. The kernel-level threads are slow and inefficient, because of the overhead associated with system calls while context-switching.

When the system uses **User-level threads**, on the other hand, they are managed entirely by the runtime environment. The kernel is agnostic to the user-level threads and treats them like single-threaded process. User-level threads are small and fast, each thread being represented by a PC, register, stack, and a small thread control block.

Merits of User-level threading

1. User-level threads can be implemented on an OS that does not support threads.
2. Unlike Kernel-level threads, User-level threads do not add clutter to the kernel code.
3. Thread switching is fast and efficient and is similar to a procedure call.
4. With User-level threads, it is possible to have multiple threads per CPU and we have full control of when they are scheduled.

Demerits of User-level threading

1. User-level threads require a thread manager that communicates to the kernel so that it can help the OS avoid making poor decisions. These poor decisions would include scheduling a process with idle threads, or blocking a process whose one thread initiated an I/O even though the process has other threads that can run.
2. Since there is a lack of coordination between the threads and the operating system kernel, the process as a whole gets one time slice whether it has five threads or five million threads. This leaves the threads to coordinate among themselves for CPU time.

3. User-level threads require a non-blocking system call, which means that the kernel should be multi-threaded. Otherwise, if for example, one thread causes a page fault, the process is blocked.

Let us look at the threading model in Golang to understand the implications of the design decision between user-level and kernel-level threading.

3.2.2 Golang's Threading Model

In the previous section, we saw that userspace scheduling is appropriate for the Golang programming model. There are three models for userspace threading.

N:1 In this approach, several userspace threads are run on one OS thread. The advantage is that context switch is seamless, but the shortcoming is that there is poor utilization with respect to multi-core machines.

1:1 In this approach, we have a one-on-one correspondence between one userspace thread and an kernel thread. In other words, every userspace thread maps onto its own OS thread. This style gives us the reverse benefits of the N:1 approach; that is, it takes advantage of all the cores on the machine, but the context switching is slow, as it has to trap through the OS via two system call overheads for every context switch.

M:N Golang takes the middle ground in this regard and uses a M:N scheduler. It schedules an arbitrary number of goroutines(M) onto an arbitrary number of OS threads(N). While this takes advantage of multi-cores and amortizes the system call overhead (since we need to pay for context switch only when they correspond to different OS threads), there is considerable complexity added to the runtime scheduler.

The runtime scheduler needs to maintain enough OS(worker) threads active so that it can use the CPU parallelism. If the runtime scheduler uses more, then parking-unparking worker threads could be a problem whereas if the runtime scheduler uses less, then it is frequently starting and stopping a new worker thread that adds to the overhead making the scheduler less power-efficient which is key in cyber-physical systems.

In summary, we can conclude that the runtime (user-level) environment can make more informed decisions about scheduling than the OS kernel for the Golang threading model because the runtime scheduler handles the interplay between goroutines, processor contexts and the worker threads.

3.3 Structures in the Golang runtime scheduler

This section covers the major abstractions and data structures used by the Golang runtime environment to perform scheduling of goroutines. In the Golang v1.8 runtime source code located at `src/runtime` in the release, the scheduler and its data structures are defined as follows:

- `proc.go` : Implementation of the Golang scheduler.
- `runtime1.go` : Parsing of the environment variables passed to the runtime environment, as well as the implementation of the tracing facility supported by the runtime environment.
- `runtime2.go` : Definition of the different data structures used by the runtime environment. This file also contains the different valid states and transitions supported by the scheduler.

There are three key structures used in the Golang scheduler - G, M, and P. A G is a goroutine, an M is a OS thread also called a worker thread in the code comments and, P is the rights and resources(also called context) required to execute the G. In order to clearly denote the three structures in diagrams, it is customary in the Golang research community to denote these with standardized notation: G is denoted by a circle, M by a triangle, and P by a square (Fig 3.2). Each of these structures maintain crucial pieces of information to execute the program, and these are indicated in Figs 3.3-3.5.

It is to be noted that there can be many Goroutines, Worker threads, and Processor Contexts in a single program. The only restriction imposed by the runtime environment is that the number of contexts is bound by the runtime

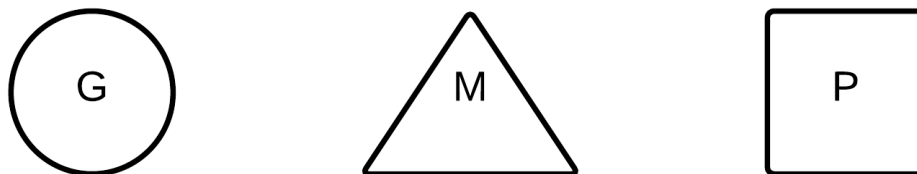


Figure 3.2: Different entities used in the Golang runtime scheduler

environment variable `runtime.GOMAXPROCS`. Typically, this is equal to the number of cores in a machine. The number of worker threads can be much higher than `runtime.GOMAXPROCS` since some of the worker threads may be blocked in a syscall. `runtime.GOMAXPROCS` does not take the blocked threads into account.

Although there are no fundamental restrictions, the maximum settable value of `runtime.GOMAXPROCS` is capped at 256.

Furthermore, the worker thread structure behaves very similarly to the standard POSIX thread that is typical in a machine running GNU/Linux. Lastly, there is no restriction on the number of goroutines.

Goroutines and Processor Contexts transition through different states as they pass through the execution cycle. The different states of a goroutine are indicated in Table 3.2 and the different states of a context structure are indicated in Table 3.3.

Finally, we have a global structure called `schedt` that maintains the overall status of the runtime environment by holding pointers to the global runnable queue, the worker threads waiting for work, the goroutines waiting to be executed, etc.

We will discuss how the scheduling of goroutines is carried out starting with the initialization of the runtime environment.

3.3.1 Initialization of the Runtime Environment and Scheduler startup

The initialization of the runtime environment is coded in `src/runtime/asm_<architecture>.s` depending on the type of architecture. For example, the

Table 3.2: Possible states of a goroutine and their definitions

Gidle	Just allocated and has not yet been initialized.
Grunnable	On a run queue. It is not currently executing user code. The stack is not owned.
Grunning	May execute user code. The stack is owned by this goroutine. It is not on a run queue. It is assigned an M and a P.
Gsyscall	Executing a system call. It is not executing user code. The stack is owned by this goroutine. It is not on a run queue. It is assigned an M.
Gwaiting	Blocked in the runtime. It is not executing user code. The stack is not owned.
Gdead	Currently unused. It may be just exited, on a free list, or just being initialized. It is not executing user code. It may or may not have a stack allocated.
Gcopystack	Goroutine's stack is being moved. It is not executing user code and is not on a run queue. The stack is owned by the goroutine that put it in <code>_Gcopystack</code> .
Gscanrunnable	GC is scanning the stack. The goroutine is on a run queue, not executing user code, and the stack is owned by the goroutine that set the <code>_Gscan</code> bit.
Gscanrunning	Same as <code>_Grunning</code> except that the goroutine is scanning its own stack
Gscansyscall	GC is scanning the stack. The goroutine is executing a system call, and the stack is owned by the goroutine that set the <code>_Gscan</code> bit. The goroutine is not on a run queue, but has an M assigned to it.
Gscanwaiting	GC is scanning the stack. The goroutine is blocked in the runtime. It is not executing user code, and the stack is not owned

Table 3.3: Possible states of a context and their implication

Pidle	The P is just allocated and has not yet been associated with an M.
Prunning	The P is assigned an M and is holding the context for a running goroutine..
Psyscall	The G associated with the P is executing a system call and is no longer using the CPU. The P can be handed over to another M to be used by it.
Pgstop	The P is at a safe point to be scanned by the GC.

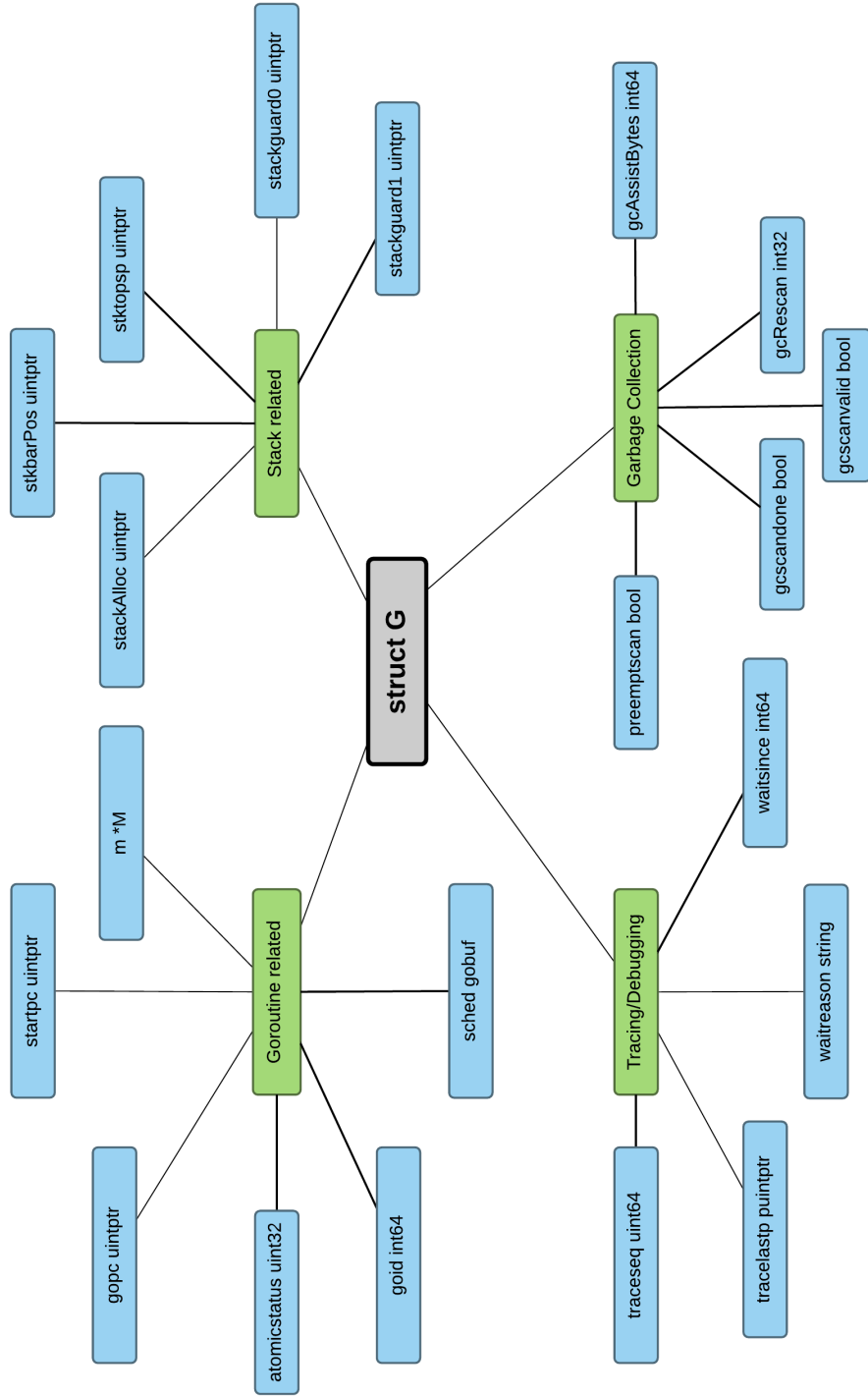


Figure 3.3: Salient fields in the goroutine structure(G)

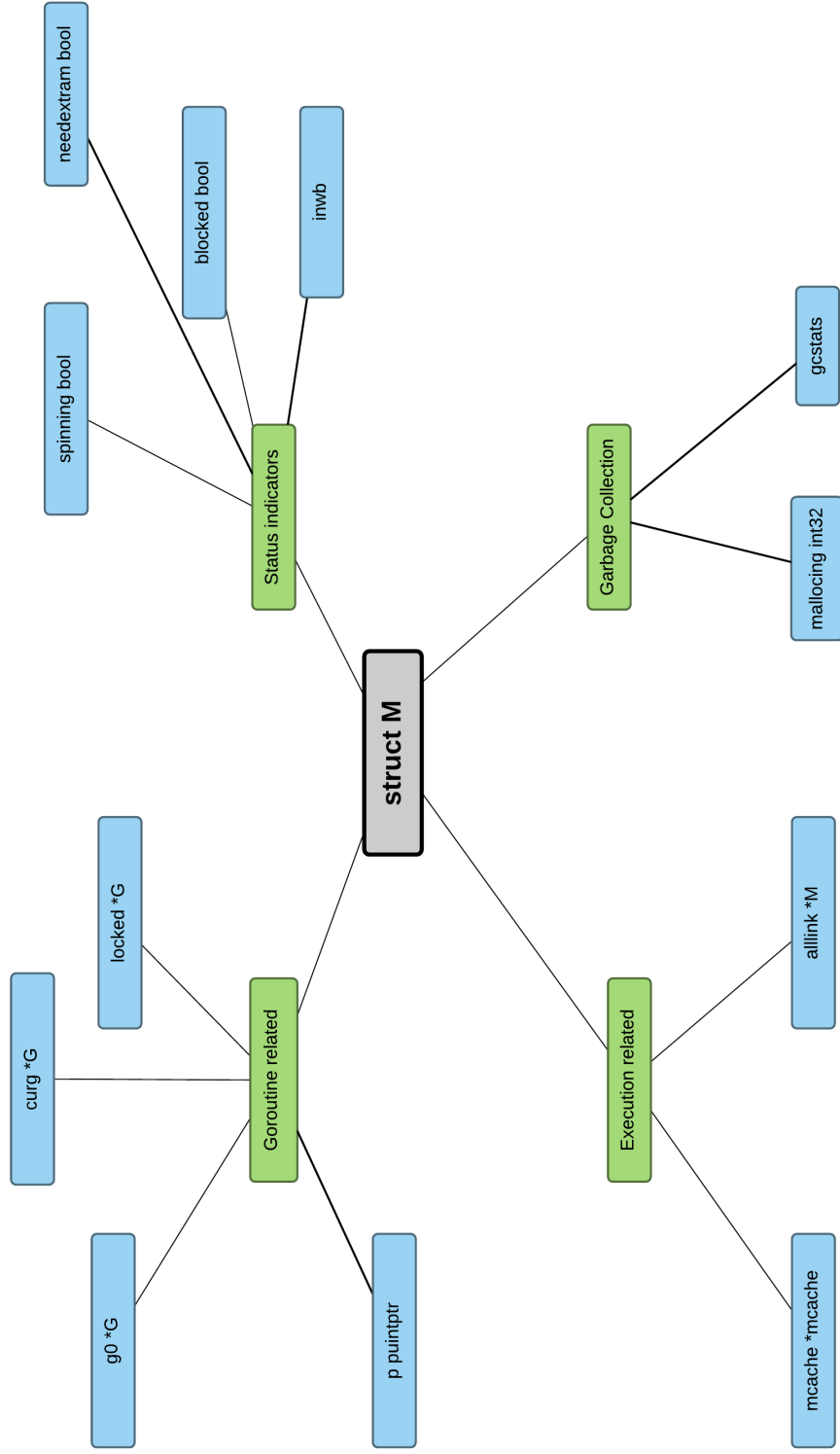


Figure 3.4: Salient fields in the worker thread structure (M)

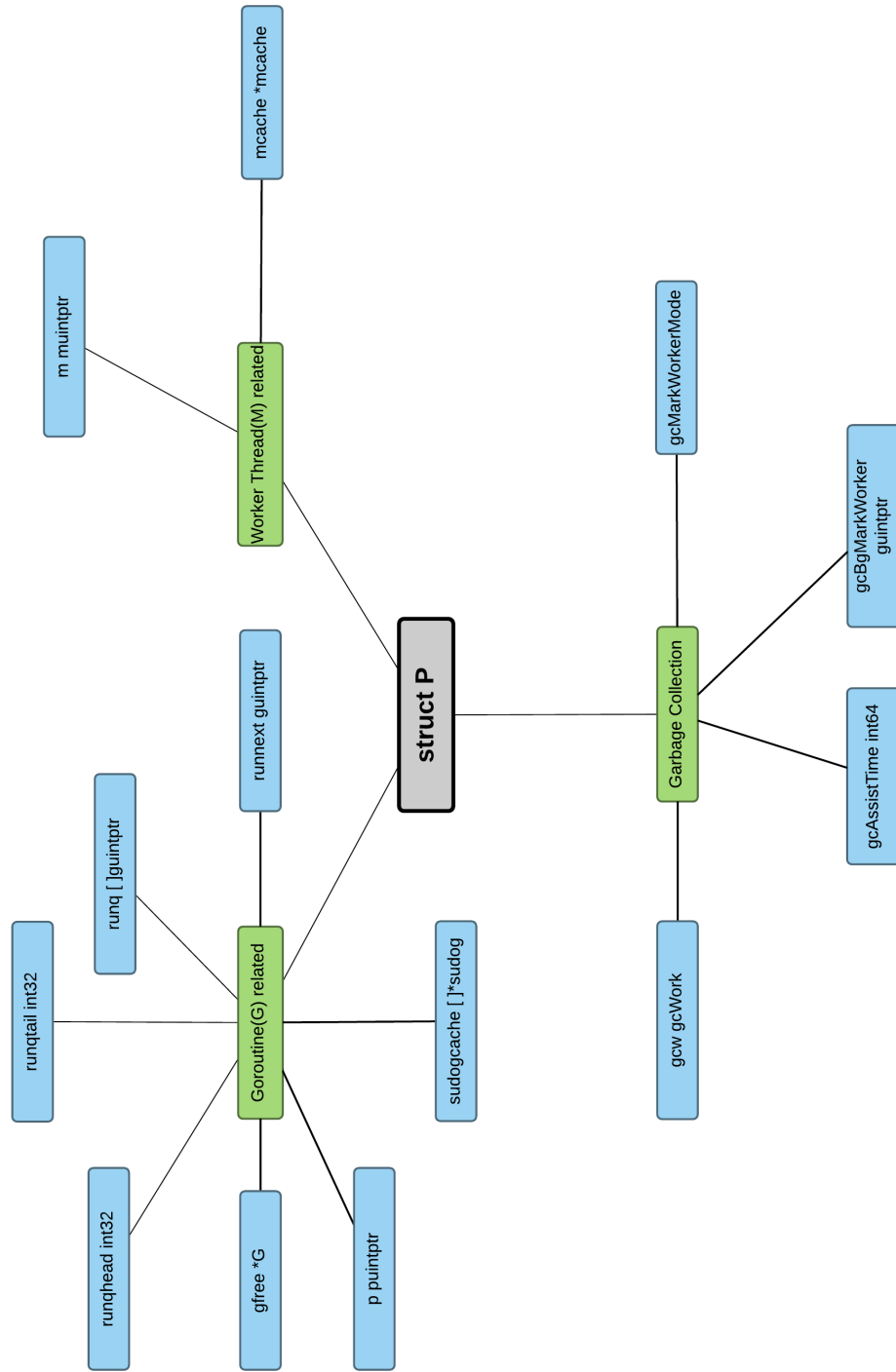


Figure 3.5: Salient fields in the context structure (P)

runtime environment initialization is in `src/runtime/asm_amd64.s` for a 64-bit CPU (Fig 3.6). The process of the OS invoking the runtime environment, *bootstrapping*, involves the following steps:

- Initially, the per-goroutine and per-machine registers are set up, which means that the thread local storage(TLS) is stored in the BX register. Then, the runtime environment performs a check to ensure the store and jump operations are successful.
- The addresses of the first goroutine (`g0`), and the first worker thread (`m0`), are stored in registers CX and AX, respectively. `g0` is a goroutine used to schedule other G's on the `m0`.
- The association of `g0` and `m0` are made by assigning `m→g0 = g0` and `g0→m = m0`. The code means that the goroutine `g0` will be running on the worker thread `m0`, and is the first association of a G struct to an M struct.
- The arguments to the runtime environment, also called `argc` and `argv`, are copied at locations `SP` and `SP+8`. The arguments are set up, and the goroutine is prepared to execute.
- Two methods in the runtime environment are called in order. The first is `osinit()`, followed by `schedinit()`. `osinit()` is used to obtain information from the OS on the physical pagesize and the number of CPUs on the machine. `schedinit()` is used to further set components in the execution environment such as the stack, the signal masks for the `g0`, malloc arena, and the garbage collector.
- A new goroutine is created to start the program given by the user, and is put on the queue of G's waiting to be run, by using the function `newproc()` in `proc.go`.
- The `mstart()` function is called, which in turn calls `schedule()`. `schedule()` is used to find a runnable goroutine using the work-stealing algorithm, which is implemented in the `findrunnable()` method. Now, the program starts

and will never return. We can say that *the main goroutine is locked onto the main OS thread*

ok:

```

// set the per-goroutine and per-mach "registers"
get_tls(BX)
LEAQ    runtime.g0(SB), CX
MOVQ    CX, g(BX)
LEAQ    runtime.m0(SB), AX

// save m->g0 = g0
MOVQ    CX, m_g0(AX)
// save m0 to g0->m
MOVQ    AX, g_m(CX)

// convention is D is always left cleared
CLD
CALL    runtime.check(SB)

// copy argc
MOVL    16(SP), AX
MOVL    AX, 0(SP)
// copy argv
MOVQ    24(SP), AX
MOVQ    AX, 8(SP)
CALL    runtime.args(SB)
CALL    runtime.osinit(SB)
CALL    runtime.schedinit(SB)

// create a new goroutine to start program
MOVQ    $runtime.mainPC(SB), AX // entry
PUSHQ   AX

```

```

    PUSHQ    $0                                // arg size
    CALL     runtime.newproc(SB)
    POPQ     AX
    POPQ     AX

    // start this M
    CALL     runtime.mstart(SB)

    MOVL     $0xf1, 0xf1 // crash
    RET

DATA       runtime.mainPC+0(SB)/8, runtime.main(SB)
GLOBL     runtime.mainPC(SB),RODATA,$8

```

Listing 3.1: Snapshot of runtime environment initialization in the Golang v1.8 source tree at `src/runtime/armamd64.s`

After the initialization process is completed, the runtime scheduler is up and running and is ready to execute goroutines.

3.3.2 Working details of the Runtime Scheduler

The previous section covered details of the major structures used in the runtime scheduler. This section will dig deeper into the scheduling process and explain the inner workings of the runtime scheduler.

Overview of the Scheduling Process

The entry point of the scheduler is the `main()` function, which sets up the runtime environment and enables the garbage collector.

At the beginning, we have a single goroutine `g0` and a single worker thread `m0` in the runtime environment. For every worker thread `M`, there is a special goroutine `g0` called the *system goroutine*. `g0` carries out the scheduling of other ‘normal’ goroutines on the worker threads. It also orchestrates the task

of creating and managing the different queues of goroutines, worker threads and processor context structures and handling their state transitions. An example is the race condition when the transition of a worker thread from spinning to non-spinning transition coincides with the submission of a new goroutine.

The lifecycle of a goroutine is as shown in Fig 3.6. Note that this is a simplified representation and does not consider interactions such as system calls, channel blocking, and pre-emption by the garbage collector.[1; 2]

The scheduling shown in Fig 3.6 is a simplistic view of scheduling, and it becomes more involved when garbage collection, system calls, and system profiling interact with the scheduler. The process of managing the different queues and handling synchronization with garbage collection makes the scheduler code very involved and long running (over 4500 lines).

The scheduler tries to maintain a dynamic equilibrium between the number of worker threads and the CPU utilization. If the number of worker threads is too few, then we are wasting the hardware parallelism; if the number is too high, then the worker threads spend a considerable portion of their lifetime parking and unparking themselves.

Goroutine, Worker Thread & Context : Scheduling perspective

This section revisits the three key structures of the scheduler with regard to the scheduling process and describes the details of their interaction.

The goroutine *G* maintains information about the goroutine. Essentially, it contains elements like program counter of the statement that created the goroutine, stack information, status (as listed in Table 3.2).The goroutine also contains a pointer to the worker thread that is assigned to execute the goroutine. This helps the scheduler to keep track of the goroutine in the runtime.

The worker thread *M* also maintains information about the current running goroutine, and the attached context structure for executing Golang code apart from signals, and profiling statistics.

The processor context *P* maintains a list of runnable goroutines, scheduler ticks, and a cache of goroutines that are free or on a waiting list. In other words,

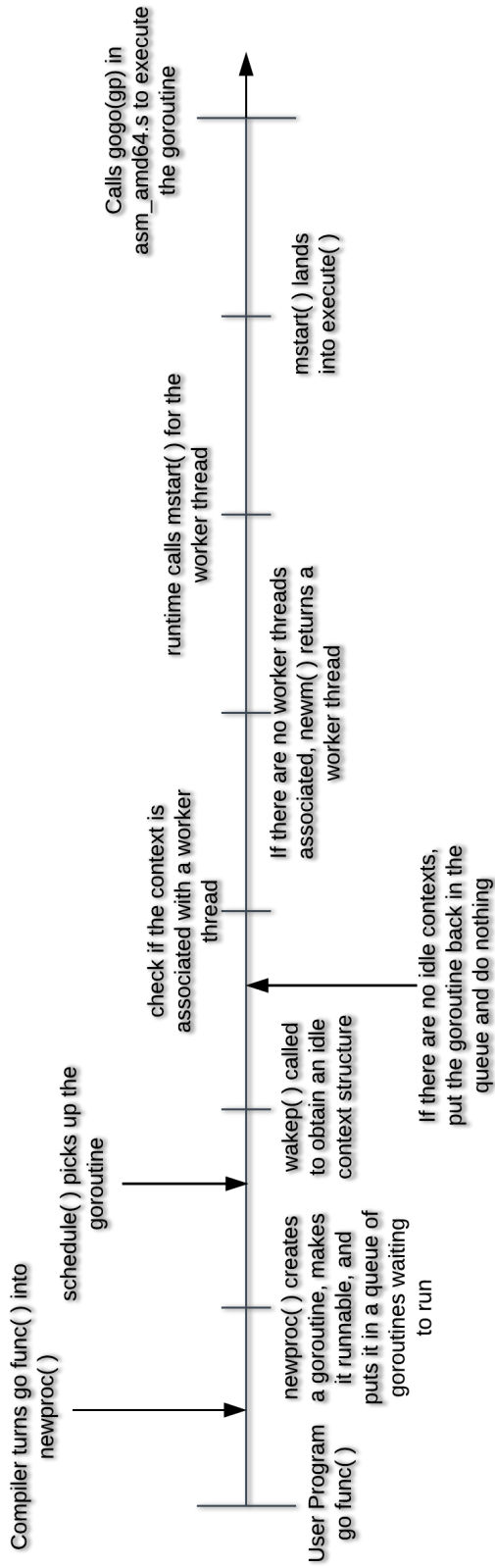


Figure 3.6: Tracing the call-graph of a goroutine as it passes from the user program to the runtime environment.

the context structure has a local list of goroutines which are waiting for CPU time. [4; 7]

Furthermore, there is also the global struct called schedt, which maintains statistics of scheduling across the system. schedt has a global list of idle worker threads waiting for work, a global runnable queue of goroutines, and another global cache of goroutines on wait lists.

Lastly, the delicate dance between these structures (while executing or waiting on a condition) comes down to the following properties:

1. M needs an associated P to execute Golang code.
2. M can be blocked or in a syscall without an associated P.
3. The scheduler state is distributed. In other words, there are per-P workqueues and a global workqueue of G's. The scheduler uses a randomized work-stealing algorithm is used to ensure that there is a fairness between the different workqueues.
4. When a goroutine becomes ready, an additional worker thread(M) is un-parked if and only if there are no spinning worker threads and there is an idle P.

Work Stealing Algorithm

The scheduler in Golang v1.8 uses a version of work stealing algorithm known as the **Randomized Work Stealing** algorithm. It is called so because this algorithm goes through per-context workqueues in a randomized fashion. The enumeration of the different P's are based on the following number theoretic relation between pseudorandomness and prime numbers:

Theorem: If a and b are coprime and i is any integer, then $(a + i) \bmod b$ is a pseudorandom sequence for different values of i.

Randomized work-stealing helps to ensure fairness among the different contexts in the runtime environment. This is shown to be provably efficient in terms of time, space, and communication for multithreaded computation.[14]

The implementation of work stealing is in the function `runqsteal()` in `src/runtime/proc.go`. Here, if one of the per-context workqueues becomes empty, it steals half of the goroutines from the local runnable queue of another context. If there is no context to steal from, the depleted context relies on the global runnable queue to obtain a batch of runnable goroutines as implemented in the function `globrunqget()` in `proc.go`.^[13]

Corner Case of Work-stealing: If there is a runnable goroutine G_b that is ready'ied by the presently running goroutine G_a , then G_b is stored in the variable `runnext` of the context struct running G_a and G_b inherits the remaining time in the timeslice, if any, that was remaining after the execution of G_a . The authors of the Golang scheduler claim this reduces the scheduling latency for a set of goroutines in the communicate-and-wait pattern.^[13]

3.4 Golang and Realtime Programming

Realtime programming requires timeliness rather than raw performance. In other words, a realtime system provides timing guarantees and also respects priority of the tasks. Realtime systems also need to respect the static priorities of the tasks which are periodically executing. When we consider Golang, there are few characteristics which make it a worthy candidate for building a realtime system as outlined below.

3.4.1 Viability of Realtime Programming Using Golang

- Userspace scheduling via the runtime uses goroutines which provide low-latency in context switching. This means if we could map realtime user functions to goroutines, we can enable low scheduling latency.
- The style of userspace scheduling in Golang uses a partially pre-emptive scheduling. Specifically, there are predefined points called *safe points*. This makes pre-emption very fast.
- Golang has mark-and-sweep garbage collector(GC). Usually, GC is known

for stopping the user program in a stop the world operation leading to missed deadlines. This can affect timing guarantees. In Golang, because of a fixed number of pre-emption points by design, the GC stops the user program during the mark phase for ≤ 2 milliseconds and Golang v1.9 aims for sub-millisecond pause time.[6]

We highlight the above points in a study of scheduling latency as compared to stock Linux Desktop and in the next chapter, we also characterize the low latency of the garbage collector in the Golang runtime.

3.4.2 Study of Scheduler Latency

One of the key aspects in scheduling is the latency. At the limit of concurrency, latency is manifested in context switches. At a high level, we can infer that if the latency of a context switch is more than the duration of the process (or goroutine), then the system cannot support so many goroutines. Finally, Realtime systems require *predictable* latency to ensure correct and consistent decisions regarding schedulability.

Characterization using perf tool in Linux

The stock Linux Desktop Ubuntu OS comes with the perf tool, which can be used to study the hardware and software performance points by extracting information from performance counters. It also provides software counters for calculations such as throughput and latency. The benchmark includes a file `sched-pipe.c`. `sched-pipe` is used to benchmark the pipe ipc(inter-process communication) mechanism in the OS and provides a means to measure context switching. The first thread writes to another thread via a pipe, which reads from the pipe. The second thread does this back and we measure time in the original thread. The time taken includes two context switches, and was measured to be around 4μseconds per operation, as shown in Fig 3.7.

```

ak7@roseline1:~/linux-repositories/linux-4.10.14/tools/perf/bench$ perf bench sched pipe -T
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

Total time: 4.092 [sec]

4.092557 usecs/op
244346 ops/sec
ak7@roseline1:~/linux-repositories/linux-4.10.14/tools/perf/bench$ █

```

Figure 3.7: Using the perf tool to measure the context switching time

Characterization the Golang runtime scheduler using channels

Since Golang uses userspace scheduling, it gives less time to context switch between two goroutines. The corresponding ipc(inter-process communication) mechanism in Golang can be simulated using the program shown in Listing 3.2. The comparison of the data is shown in Table 3.4.

```

package main
import (
    "fmt"
    "testing"
)
func main(){
    fmt.Println("sync", testing.Benchmark(
        BenchmarkChannelSync).String())
    fmt.Println("buffered", testing.Benchmark(
        BenchmarkChannelBuffered).String())
}
func BenchmarkChannelSync(b *testing.B){
    ch := make(chan int)
    go func(){
        for i := 0; i < b.N; i++ {
            ch <- i
        }
        close(ch)
    }
}

```



```

    }()

    for _ = range ch{
    }

}

func BenchmarkChannelBuffered(b *testing.B){
    ch := make(chan int, 128)
    go func(){
        for i := 0; i < b.N; i++ {
            ch <- i
        }
        close(ch)
    }()

    for _ = range ch{
    }

}

```

Listing 3.2: Two goroutines communicating with one another for measuring context switching time. [8]

Few observations follow from Table 3.4:

1. The context switching time via the kernel scheduler is approximately 8-10x times worse compared to userspace scheduling.
2. As expected, the context switching time for unbuffered channels is lower compared to buffered channels because unbuffered channels block goroutine to be ready to send or receive whereas, buffered channels are non-blocking and return immediately, exhibiting lesser switching time.

Table 3.4: Context Switching Time (Scheduling Latency): Golang runtime Vs. Linux

Linux OS with pthreads	4090 nanoseconds
Go runtime with Buffered Channels	265-350 nanoseconds
Go runtime with Unbuffered Channels	735-800 nanoseconds

3.5 Realtime Golang

The runtime environment, which runs the work-stealing algorithm, is the focal point where we integrate the realtime features into the system. This setup is advantageous because:

1. The runtime layer, being higher up than the OS in abstraction, has the advantage of being able to relay the timing information to the OS, while offloading some of the tasks such as memory management (garbage collection) from the OS.

2. The changes in the runtime does not require recompilation of the OS every time we need to change a mechanism or a policy parameter. The installation of the runtime environment is considerably simpler compared to a fresh kernel, and the compilation takes under 30 seconds, as noted in Chapter 2.

The changes to the runtime environment are centered around adding the realtime ability to a goroutine. To this end, we modify the goroutine struct to include deadline, worst-case execution, priority, and period. This modification also requires changes while switching between goroutines, and in inter-process communication using channels. Realtime Golang requires changes to the following files in the runtime environment:

- runtime2.go
- proc.go
- select.go

In runtime2.go, the definitions for the different scheduler structures G, M, and P are present. First, in a goroutine, we add four parameters - deadline, worst-case execution time, priority, and period. These four parameters give the goroutine the realtime characteristics. For simplicity, we have assumed deadline

to be same as the period and have assumed all the realtime goroutines to have the same priority. The observations without priority can be extended to priority-enabled realtime scheduling as well. Furthermore, we also add two fields - `isrt` bool, and `scheduler_time` int64, the former is used to indicate if the goroutine under consideration is a realtime goroutine. Second, in the worker thread, we add a field `iscurgrt` to indicate if the worker thread is presently executing a realtime goroutine. Next, in `P`, we add special queues for processing realtime goroutines and also a cache of realtime goroutine ids. Finally, in the `schedt` structure as well, we add a global queue of runnable realtime goroutines.

`proc.go` contains the implementation of the work-stealing scheduler. The scheduler's job is to match up a goroutine (the code to execute), a worker thread (where to execute it), and a context (the rights and resources to execute it). When a worker stops executing user Golang code, for example by entering a system call, it returns its context to the idle context pool. In order to resume executing user Golang code, for example on return from a system call, it must acquire a context from the idle pool. All `G`, `M`, and `P` objects are heap allocated, but are never freed, so their memory remains type stable.[13]

Paths affecting timing performance

There are 3 possibilities for a goroutine to block. These have the maximum impact on the timing performance of Golang.

System Calls such as file or disk I/O: When a goroutine(`G`) invokes a system call, it does not need the context (`P`) and is waiting on the operating system worker thread(`M`). So, the goroutine releases the context, and remains running on the worker thread. The state of the goroutine is changed to `_Gsyscall`. In the runtime, there is a background thread (another `M`), called `sysmon()` that checks for runnable goroutine and available contexts. It does this on a periodic polling period of 20 μ s to 10ms. When `sysmon()` sees this criterion being met, it wakes up a sleeping worker thread or starts a fresh worker thread and associates the goroutine and context structure with this.

When the system call is finished, the goroutine needs the context structure

back. So it reacquires the context structure. If a context is not available, the goroutine is marked as runnable and added to the global workqueue (then the work-stealing to kick in when a context looks in the global workqueue for work).

Unbuffered channel Operation Unbuffered channel operations are blocking. For a sender goroutine needs a receiver goroutine to receive the value through the channel for it to finish and vice versa. When a goroutine is blocked on a channel operation such as waiting for the send or receive to complete, the worker thread looks for a runnable goroutine, since the blocked goroutine is not runnable now. If the worker thread is not able to find a runnable goroutine, it releases the context and goes back to sleep. If worker thread is able to find a runnable goroutine, it runs the goroutine (as it already has a context).

When the channel operation finished, the original goroutine which was blocked is now runnable. The channel checks for available contexts (since a blocked goroutine lose its context structure). If there are available contexts, the channel wakes up a worker thread to run the goroutine. If there are no contexts, the runnable goroutine is again added to the global workqueue.

Garbage Collection Stop the World The GC stops the user program when it is in its mark phase. During this (short) time, the GC stops all the running goroutines and puts all the worker threads to sleep. When GC's mark termination phase is over, it waked up all the worker threads. The worker threads find a runnable goroutine and if it manages to find one, acquires a context and continues program execution.

Modification to Queues

In Golang realtime, we use a list to implement a heap data structure. The min-heap is used to sort the pushed goroutine according to the deadline parameter when a fresh goroutine is added to the list or removed from the list. This implementation draws inspiration from the timer infrastructure in Linux OS. In the realtime heaps, we implement the earliest deadline first algorithm similar to how timers are implemented in Linux. The heaps are arranged according to a min-heap criterion based on the deadline of the realtime goroutine.

Change in runqget()

runqget() is a function used to get a goroutine from a local runnable queue. In Realtime Golang, we check for any goroutines in the realtime heaps prior to checking the local runnable queues. When a goroutine G_b succeeds G_a , the inheritance of the remaining timeslice from G_a is carried out in an identical fashion to the stock implementation of the scheduler.

Realtime Work-stealing

runqgrab and runqsteal are the central functions implementing the work-stealing algorithm over the per-context workqueues. In fact, runqsteal uses runqgrab under the hood to grab half of the routines. The modified versions of runqgrab and runqsteal uses the per-context realtime queues against the per-context goroutine queues. Likewise, we also perform similar changes to globrunqputbatch, globrunqget, runqput, and runqslow.

Modules in Golang vs. Realtime Golang

Module	Standard Golang	Realtime Golang
Compiler	Default gc	Same as Standard Golang
Runtime Scheduler	Work-Stealing Scheduler	Work-Stealing Scheduler with Realtime queues
Runtime Structures	Default G, M, P, and schedt	G, M, P, and schedt with Realtime fields added
Supports Realtime Properties for User Program	No	Yes
Garbage Collector	Concurrent Mark & Sweep	Same as Standard Golang

Issues while tracking goroutines

While programming user goroutines(G) with realtime parameters is just a minor change to the runtime environment and the user program, we are presently

facing issues with goroutine management in the runtime, partly because in addition to user goroutines, the runtime maintains additional goroutines (called g0) associated with each of the OS threads (M).

It looks like the goroutine for which we programmed the params is either exiting or being blocked in a syscall or there is thread (M) migration happening and while starting a new thread, work-stealing is triggered leading to us losing state of the goroutines. This is currently under debug and for now, we are setting the parameters manually in the runtime to overcome this issue.

3.6 Conclusion

This chapter outlines the userspace scheduler used in the Golang programming language and the different tools used to benchmark the Golang runtime scheduler. It also gives an overview of the changes in the runtime which can be used to provide realtime guarantees.

An outline of the changes required for making Golang realtime are:

- In each goroutine, we add 4 realtime fields: Deadline, Worst-case Execution Time, Period, and Priority. and a variable to track the time a goroutine has been in running state in the scheduler.
- In each worker thread, we add a boolean field to indicate when the currently executing goroutine on it is realtime.
- In each context structure, we add 3 separate queues to maintain realtime goroutines: One is the queue of runnable goroutines, second queue of realtime goroutines on a waiting list, and a third queue of goroutines which are dead.
- Add the realtime implementation of `runqgrab` and `runqsteal`, which deal with realtime runqueues.
- For every submission of realtime goroutine to the list of goroutines in `allgadd`, we arrange the goroutines according to the deadline using heap sort

algorithms. We do the same when a goroutine is removed from the list.

- For select timeout implementation, on the channel receive, we override the timeout when there are realtime goroutines in the queue and the normal goroutine queue is empty. Similar idea is used on the send side as well.
- Lastly, we have modified the goroutine submission and readying functions to include realtime queue as well.
- Finally, we have a function that the user program uses to set realtime parameters into the runtime. This requires changing the user program to call this function.

Finally, we note that the development is still a work in progress, and we can use the runtime scheduler changes as a starting point to develop Golang as a realtime programming language.

Bibliography

- [1] Morsing, Daniel. "The Go Scheduler." The Go Scheduler. N.p., 30 June 2013. Web. 15 Apr. 2017, <https://morsmachine.dk/go-scheduler>
- [2] Wang, Bin. "Notes On Go Scheduler." N.p., Sept. 2014. Web. May 2017, <http://www.binwang.me/2014-09-01-notes-on-go-scheduler.html>
- [3] Jansen, Jean Bernard. "The Low Level Awesomeness of Go." Slideshare. N.p., 31 Oct. 2016. Web. 15 May 2017, <https://www.slideshare.net/JeanBernardJansen/the-low-level-awesomeness-of-go>
- [4] Fang, Zhou, Mulong Luo, Fatima M. Anwar, Hao Zhuang, and Rajesh Gupta. "Go-RealTime: A Lightweight Framework for Multiprocessor Real-Time System in User Space." 4th IEEE International Workshop on Real-Time Computing and Distributed Systems in Emerging Applications (2016): n. pag. Web. 15 May 2017
- [5] "Concurrency Is Not Parallelism." The Go Blog. N.p., 16 Jan. 2013. Web. 15 Apr. 2017, <https://blog.golang.org/concurrency-is-not-parallelism>
- [6] "Package Runtime." Runtime - The Go Programming Language. N.p., n.d. Web. 15 Apr. 2017, <https://golang.org/pkg/runtime/>
- [7] Deshpande, Neil, Erica Sponsler, and Nathaniel Weiss. "Analysis of the Go runtime scheduler." (2011): n. pag. 12 Dec. 2011. Web. 15 Apr. 2017.
- [8] Donovan, Alan A. A., and Brian W. Kernighan. The Go Programming Language. New York: Addison-Wesley, 2016. Print.
- [9] Love, Robert. Linux Kernel Development. Upper Saddle River: Addison-Wesley, 2015. Print.

- [10] Golang UK Conference 2016 - Dave Cheney - Seven Ways to Profile Go Applications. Dave Cheney. YouTube. N.p., 8 Sept. 2016. Web. 15 May 2017, https://www.youtube.com/watch?v=2h_NFBFrciI
- [11] DotGo 2016 - Rhys Hiltner - Go's Execution Tracer. Dir. Rhys Hiltner. YouTube. N.p., 17 Jan. 2017. Web. 15 May 2017, https://www.youtube.com/watch?v=mmqDlbWk_XA
- [12] Introduction to Go Tool Trace. Dir. Pusher. YouTube. N.p., 6 Apr. 2017. Web. 15 May 2017, <https://www.youtube.com/watch?v=Xq5HDH8y0CE>
- [13] "Google Go Source Code." Google Git. Google, Feb. 2017. Web. July 2017. <https://go.googlesource.com/go/+go1.8>
- [14] Blumofe, Robert D., and Charles E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing." *Journal of the ACM (JACM)*. ACM, Sept. 1999. Web. 19 Mar. 2017. <http://dl.acm.org/citation.cfm?id=324234>

Chapter 4

The Golang Garbage Collector (GC)

Garbage collection is widely touted as the bane of realtime computing. This chapter explains the garbage collector (GC) used in the Golang runtime environment, and studies the timing performance of the GC used in Golang v1.8 to benchmark its viability for realtime programming.

The key idea in realtime garbage collection is to trade throughput for latency. In such a case, the original program may be allowed to run slower, but only to a negligible extent, at the expense of providing a timing guarantee.

4.1 Overview of Garbage Collection

Garbage collection is the way of optimizing the amount of memory the program uses. In particular, garbage collection is the process of removing the objects that the program does not need at the moment.

In general, when any program is compiled and run, it is loaded onto the primary memory (also called RAM). In the RAM, the memory footprint of the program can be split into several regions, each of which serve a particular purpose. Typically, a program's memory usage can be split into the following regions: code segment, data segment, heap, and the stack. The code segment contains the compiled machine-code of the program. The code segment is typically read-only. The data segment contains the data used by the program, and can be modified. The data segment may be thought of as consisting of two

sub-segments: the BSS, which contains uninitialized objects declared outside functions as well as uninitialized static local variables; and the ESS, which contains the initialized objects in a program. The heap is used by the runtime for dynamic memory management, which is typically used by the Golang runtime layer to allocate and deallocate space transparently. The stack, like with other languages, contains local variables and function arguments. Similar to the heap, the stack changes in size over the course of execution of the program.

4.2 Garbage Collection Internals

Golang's GC is based on the tricolor algorithm by Dijkstra circa 1978.[1] The GC is concurrent and performs according to the mark-sweep algorithm. Dijkstra called the running program a mutator, since the program changed the state of the memory for the GC.

The GC views a heap as a connected graph of objects. The objects are either white, grey or black - the three colors of the tri-color algorithm as described in the table below:

Color	Definition	Garbage Collected?
White	Set of objects that are going to be garbage collected	Yes
Black	Objects which are i) reachable from the root, and ii) having no outgoing references to any white object	No
Grey	Objects which are i) reachable from the root, and ii) yet to be scanned to check for references to white objects	No

At the start of a garbage collection cycle, all of the objects are marked white. Then, the GC scans for accessible items, such as globals and local variables, and marks them grey. If the GC finds a new reachable white object during the second scan, it turns the white object into grey. Finally, the GC chooses a grey object, blackens it, and then scans it for pointers to other objects. [9] This

is done till the there are no more grey objects. At this point, unreachable white objects and their corresponding memory may be reused for other purposes.

The GC and the running program maintain the invariant that black objects in the heap never hold a reference to white objects. Furthermore, the state of the color of an object changes from white to grey and grey to black. This rule is ensured by the runtime by using a write barrier, which makes sure that any changes made by the mutator is 'seen' by the GC.

4.3 Memory Allocation in Golang runtime

Golang's garbage collection was *originally* based on the TCMalloc tool.[8] TCMalloc was so called because it assigned each thread a thread-local cache, and small allocations, less than 32 kilobytes, were made from these thread-local caches as needed. TCMalloc is a refinement of glibc's malloc, which was also called ptmalloc2. Large allocations, on the other hand, were directly taken from the central heap using a page-level allocator, a page being 4K-aligned region of memory. Periodically, garbage collections were used to migrate memory back from a thread-local cache into the central data structures.

4.3.1 Advantages of TCMalloc over standard malloc

Golang was designed for speed and concurrency, and thus, the designers of Golang were not content with the speed of standard malloc.

- **Speed:** glibc's malloc took 300 nanoseconds to execute a malloc/free on a 2.8GHz processor, whereas the TCMalloc took a meagre 50 nanoseconds for the same set of operations.[8]
- **Memory Usage:** TCMalloc also reduces lock contention for multi-threaded programs. For small objects, there is virtually zero contention. For large objects, TCMalloc tries to use fine grained and efficient spinlocks. ptmalloc2 also reduces lock contention by using per-thread arenas, but there is a big problem with ptmalloc2's use of per-thread arenas. In ptmalloc2,

memory can never move from one arena to another. This can lead to huge amounts of wasted space.

In Golang, memory allocation has diverged quite a bit from TCMalloc, but the notion is kept essentially the same. Memory allocation works in runs of pages, each page being 8kB in size, and allocations can be done in multiples of page sizes as well. Small allocation sizes, up to and including 32kB, are rounded to one of about 70 size classes, each of which has its own free set of objects of exactly that size. Any free page of memory can be split up into a set of objects of one size class, which is then managed using a free bitmap.[3]

4.3.2 Data Structures Terminology in Memory Allocation

The allocator uses the data structures, which are defined as follows:

- **fixalloc** is a free-list allocator for fixed-size off-heap objects, used to manage storage used by the allocator.
- **mheap** is the malloc heap which is managed at page granularity.
- **mspan** is a run of pages managed by the mheap.
- **mcentral** collects all spans of a given size class.
- **mstats** maintains the allocation statistics of the memory.
- **macache** the cache of worker threads maintained by each context structure.

4.4 Analyzing the Garbage Collection in Golang

We study the timing characteristics of Garbage Collection in Golang to understand the implication of having garbage collection in a realtime setting. We do this by considering two benchmarks - one is a large binary tree traversal, and another is a package parser for Golang. In the first case, we see the GC pause time is well-within the realtime realm whereas in the second case, the GC pause time shows much higher variance.

4.4.1 Binary Tree Benchmark

The benchmark consists of generating a balanced binary tree of depth 16, and traverse them starting from a depth of 4, going all the way till the maximum depth of 16.[7] At each element, the value of a tree is calculated. The value of a tree is defined as the algebraic recursive sum of (value of root + value of left child - value of right child). Clearly, in this case, the tree can be garbage collected at lower depths before going onto the deeper sections of the tree. The value of a tree at a higher depth does not require the lower depths and those can be garbage collected. Naturally, the collector will be triggered to satisfy allocation requests. The go tool trace output of this tree generator is shown in Fig 4.1.

We see that for the vast majority of time, the main goroutine is running on Proc0, whereas when the GC is triggered, the Proc0 switches to running the garbage collector goroutine. The GC cycle lasts for slightly over 2 milliseconds, which is acceptable to most of the realtime applications.

4.4.2 Golang Package Parser

Another example of garbage collection is parsing the list of Golang packages repeatedly.[3] This is a more generalized version of the first benchmark. In the loop parsing a directory, non-overlapping directories can be garbage collected and the unused objects would be discovered at the end of the mark phase. The trace output for this is shown in Fig 4.2.

As expected, the garbage collection time here varies, since the size of garbage at each point varies. For instance, while collecting garbage from directories containing more number of files, it is expected to take more time compared to collecting garbage from directories containing fewer files. The garbage collection pause times in this case range from 5 milliseconds to 130 milliseconds with a standard deviation of approximately 35 milliseconds. This is a higher number than what we were hypothesizing but gives us a measure of the latency we can expect to see in such a usecase.

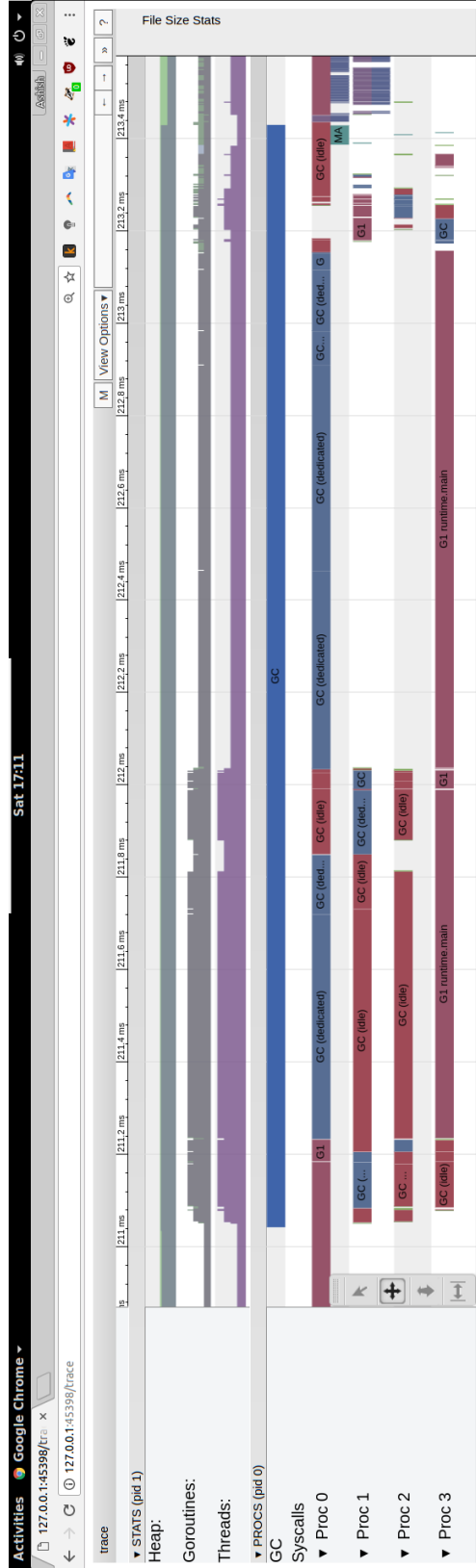


Figure 4.1: Trace output of the `go tool trace` command for the Binary Tree Benchmark showing pause times of 2 milliseconds.

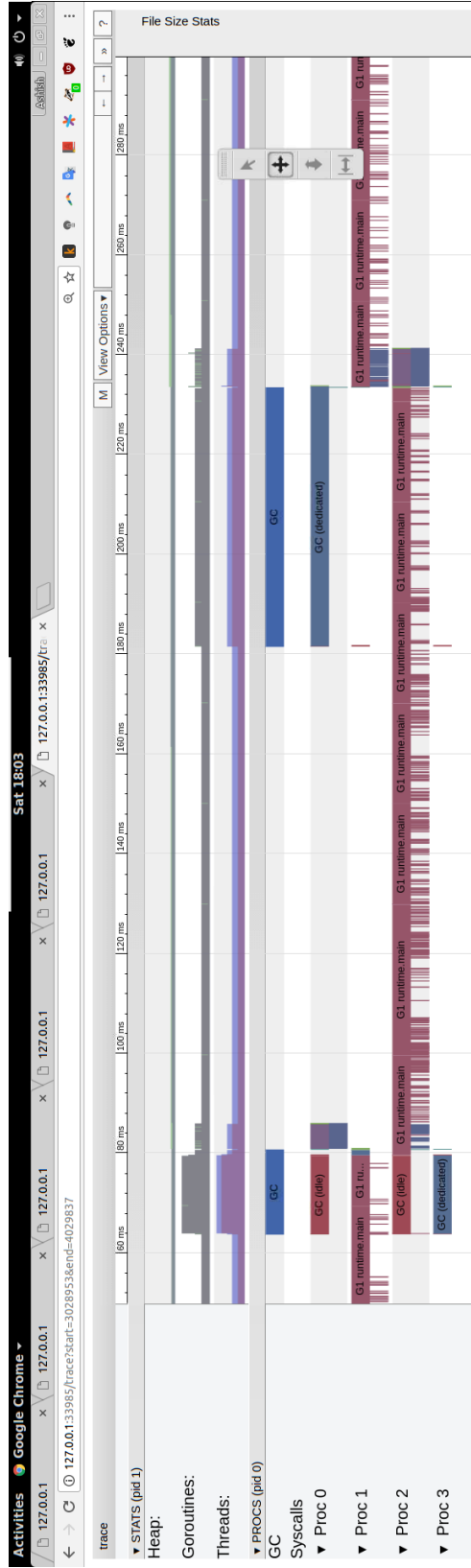


Figure 4.2: Trace output of the `go tool` trace command for the Go Parser package showing variable pause times.

4.5 Conclusion

In this chapter, we went through the garbage collection mechanism used in Golang and studied the timing characteristics of the runtime garbage collector to correlate it to timing information in Chapter 3. We see that from the timing analysis of the garbage collector that it is inconclusive for it to be used directly in a realtime scenario.

Bibliography

- [1] Dijkstra, Edsger W. "On-the-fly Garbage Collection: An Exercise in Cooperation." *Communications of the ACM*, Nov. 1978. Web. 15 Apr. 2017, <http://dl.acm.org/citation.cfm?id=359655>
- [2] Jim, and Will. "Golang's Realtime Garbage Collector." *Sessions by Pusher*. Pusher Inc., Feb. 2017. Web. 1 May 2017, <https://pusher.com/sessions/meetup/the-realtime-guild/golangs-realtime-garbage-collector>
- [3] "Google Go Source Code." *Google Git*. Google, 16 Feb. 2017. Web. 13 July 2017. <https://go.googlesource.com/go/+go1.8>
- [4] "Go 1.5 Concurrent Garbage Collector Pacing." *Go Garbage Collector*. Google, 03 Oct. 2015. Web. 15 May 2017, <https://golang.org/s/go15gcpacing>
- [5] "Go 1.5 Release Notes." *Go 1.5 Release Notes - The Go Programming Language*. Google, Aug. 2015. Web. May 2017, <https://golang.org/doc/go1.5gc>
- [6] Jones, Richard, and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Chichester: John Wiley, 2007. Print.
- [7] Troestler, Christophe, and Einar Karttunen. "The Computer LanguageBenchmarks Game." *Binary-trees Description (64-bit Ubuntu Quad Core) | Computer Language Benchmarks Game*. N.p., Web.May 2017. <http://benchmarksgame.alioth.debian.org/u64q/binarytrees-description.html>
- [8] Ghemawat, Sanjay. "TCMalloc : Thread-Caching Malloc." *TCMalloc : Thread-Caching Malloc*. Google, 15 Apr. 2017. Web. 15 July 2017, <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>
- [9] "Tracing Garbage Collection." *Wikipedia*. Wikimedia Foundation, 02 July 2017. Web. 20 July 2017. https://en.wikipedia.org/wiki/Tracing_garbage_collection

Chapter 5

Conclusion

This thesis provides an outline of the strategy to formulate Golang as a realtime programming language by studying the timing characteristics of the scheduler and garbage collector. We analyzed the different paths of the scheduler which are critical to timing and propose a software patch on the runtime scheduler. This approach is aimed at saving the user from the task of migrating to a realtime operating system to run realtime tasks.

We used this patched runtime scheduler along with the concurrent garbage collector of Golang v1.8 to benchmark garbage collection times.