CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

RECREATING TOP USING GOLANG

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

by

Daniel Hunt

December 2018

The thesis of Daniel Hunt is approved:

Robert Mcilhenny, Ph.D.	Date
Richard Covington, Ph.D.	Date
Jeff Wiegley, Ph.D., Chair	Date

California State University, Northridge

Table of Contents

ABSTRACT

RECREATING TOP USING GOLANG

By

Daniel Hunt

Master of Science in Computer Science

Writing system tools for Linux systems can be time consuming to develop and maintain. The paper explores recreating the popular tool called Table Of Processes (a.k.a. TOP) using the Go language. It also compares developing tools in Golang versus the C language. Communities of developers are creating vital Linux tools in Golang and Rust, proving it may be time to consider other options.

Chapter 1

Introduction

A common suggestion for beginner developers is to contribute to open source projects. While there may be many out there, contributing to any of the important system tools that run on linux systems can be very intimidating. Many of the important tools on linux were written in C (i.e., Table Of Processes) and have been around for some time. Dealing with mature C code is definitely no easy task, which has led communities of developers to thinking there must be an easier way to develop and maintain tools on Linux. This paper explores an alternative to developing system tools in the Golang language.

Today's society has seen a massive improvement in computing. This has allowed developers to create programming languages more suitable for certain tasks. The Go language for example, was developed at Google to resolve many problems criticized by today's developers. While it certainly isn't a replacement for C, many think it should be one of the first choices when beginning a new software project.

For example, let's take a look at the Table Of Processes (TOP) command. This tool's purpose is to display information about the system being used. Some pieces of information it displays is the memory and swap usage. Perhaps one of the most important pieces of information it displays is the table of processes running on the system sorted by CPU or Memory usage. Without knowing how the tool works internally, it seems like it may be doing all kinds of special tricks to get that kind of system information. When taking a closer look, one will see that from a higher level perspective, it simply scans different areas of the process filesystem to acquire that information. The process filesystem (a.k.a. /proc), is a folder in the root directory that contains multiple numbered (process ID) folders for each of the different processes. Each numbered folder represents a process ID and contains information about said process. In addition to reading in the information, it makes some calculations for each process and displays it in a more human readable format. It does all that over a certain time interval, making sure the user is always aware of what is currently going on in the system.

When taking a look at the code for `top` or even `htop` (a nicer looking version of `top`), the code was written in C some time ago. Yet, many developers today still use this tool on a regular basis for a variety of reasons. The C language still remains popular for many low level computing tasks that require speed. Should we have developed the tools `top` and `htop` in C if we are just reading the process filesystem over a certain interval (i.e. 1 second)? Back when it was developed, there weren't any serious languages to compete with C for this kind of task. That has changed today with Golang being one of the most popular and supported languages on Github.

In order to get a better understanding of system tool development, this paper discusses how to recreate a popular tool (TOP) on Linux using Golang (now referred to as GTOP). Recreating this popular tool requires a better understanding of the proc filesystem and the programming language Golang. Using Golang to accomplish the task may seem fun and easier, using C may still be a better choice for other projects.

Chapter 2

Diving into the process filesystem

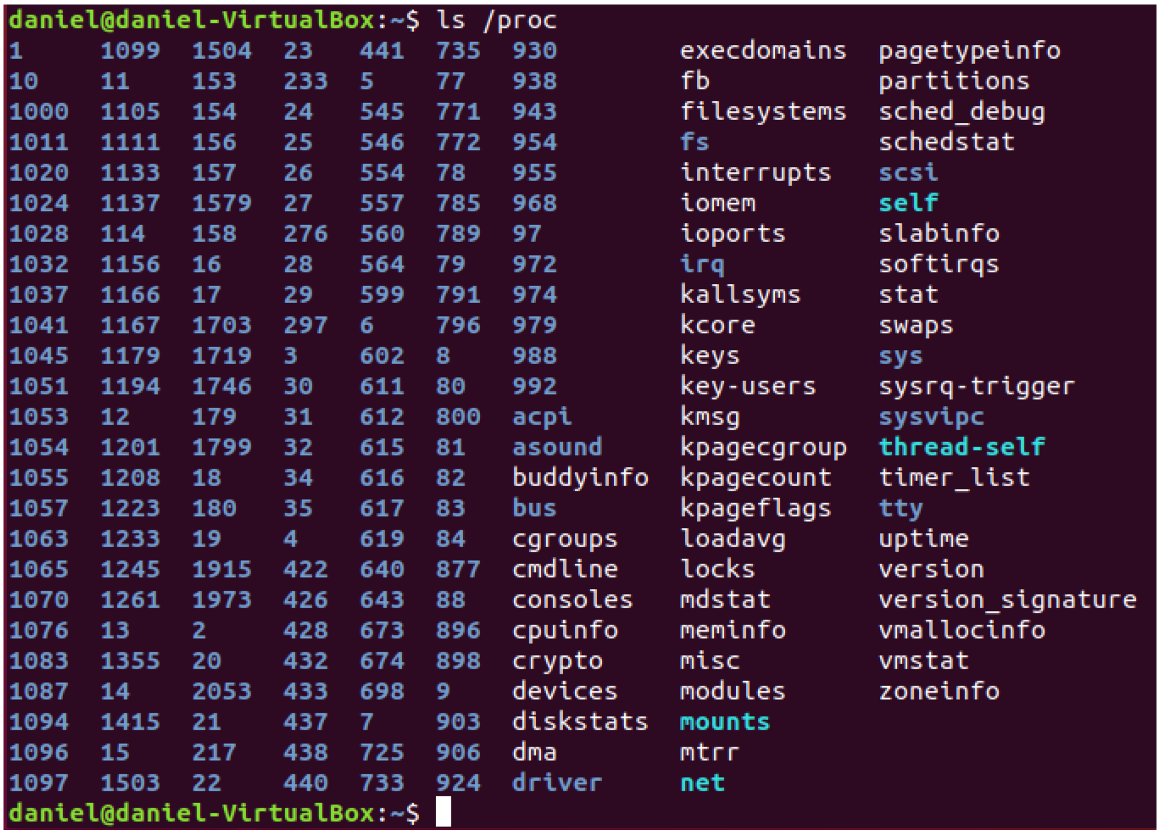

Figure 2.1: Output of ls /proc

2.1 What is the process filesystem?

If one were to list all the directories and files in the process filesystem, it would look like any normal folder on the system. It has sub-folders (most of them numbered) and files. An interesting fact about process filesystem, it is often referred to as a psuedo-filesystem or virtual filesytem. This means the files and folders shown in the process filesystem are not actual files stored on disk. The data shown in figure 2.1 is actually runtime files (or virtual files). These folders and files in the process filesystem contain and control information about the system kernel. Not many developers understand the importance of the process filesystem, their view changes when they learn they can modify the kernel parameters while the system is running. This file system will be very important to developing a Golang version of `top`.

```
daniel@daniel-VirtualBox:~$ ls -l -S /proc | head -n 25
total 0
-r--------  1 root            root            140737477881856 Sep 22 11:26 kcore
lrwxrwxrwx  1 root            root                         11 Sep 22 11:26 mounts -> self/mounts
lrwxrwxrwx  1 root            root                          8 Sep 22 11:26 net -> self/net
dr-xr-xr-x  9 root            root                          0 Sep 22 11:11 1
dr-xr-xr-x  9 root            root                          0 Sep 22 11:11 10
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1000
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1011
dr-xr-xr-x  9 root            root                          0 Sep 22 11:11 1020
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1024
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1028
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1032
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1037
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1041
dr-xr-xr-x  9 root            root                          0 Sep 22 11:11 1045
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1051
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1053
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1054
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1055
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1057
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1063
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1065
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1070
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1076
dr-xr-xr-x  9 daniel          daniel                        0 Sep 22 11:11 1083
daniel@daniel-VirtualBox:~$
```

Figure 2.2: Output of ls /proc that shows the sizes of the folders.

Figure 2.2 shows a different version of the output from the `ls` command. The column to the left of the date represents the file size stored on the system in bytes. The first row says total and is the total size the directory takes up on disk; zero because these files are virtual. The second row is a core system file which maps directly to every single byte in the system. The third row is a symlink to a mount for this filesystem. Notice the first row says the size is zero, yet the second and third row are non-zero numbers. Figure 2.1 doesn't actually say the second and third row is the in the process filesystem file system. This shows that any of the files stated to be in the process filesystem, aren't actually taking up space on the disk.

2.2 Process file system information

The process file system is also know as `/proc` or `proc`. The previous section said the `proc` file system contains system parameter information and takes up no space on disk. So where is the process and system information needed for the `top` tool? Figures 2.1 and 2.2 both show there are numbered directories in the process file system. Those numbers represent the process identification number (a.k.a. PID) of any process on the system.

Since those numbers in the process file system are directories, we can look inside and see there are many more files and folders in there. For this paper, the stat and status files are the files that will be analyzed. The status file is used to get the owner and memory information of the process. The stat file is used to calculate CPU usage of the process.

The last file that is read is the file named `meminfo` (memory information of the system), that is at the top of the process file system (`/proc/meminfo`). The main purpose of reading from here is to get the memory information of the entire system and not just the usage of one process. Therefore, it is the root of the process file system since it has no direct relation to any of the processes.

Chapter 3

Linux TOP

3.1 Structure

The `top` command is a tool to view the list of processes on a system sorted by memory and CPU usage. In addition, it also displays system hardware information (memory, swap and buffer usage). As described earlier, it is basically a scanner of the process filesystem that displays information in a nicely formatted way. This makes it easy to split up the code for the tool. The two main parts to this tool is the system hardware and system process information. The command iterates over a certain interval, one second will be the time interval used for this paper. It should be noted there is nothing wrong with the tool, it is only being used for demonstration purposes. The goal is to show there are more efficient and less complicated ways of building system tools using other languages such as Golang. The `top` tool makes a great example because most of the operations are reading and iterating over files and directories. There is nothing computationally expensive about it that requires the use of the C language. The algorithm for this tool is simple, just continue to loop while scanning for hardware information and process information. After scanning for the information, do the calculations, display and wait one second. In terms of scanning, organizing the information properly is crucial to making it easier to develop. In this case, creating a structure to store processes information is needed. It becomes easier to store all the process and their information in an array of those structures. There is obviously a little more too it than that. Most of it is details as to how it displays the information properly.

Chapter 4

Golang Primer

4.1 About Golang

Golang is a programming language loosely based on the C language. It was developed at Google with the intention to create a language without all the bloat and confusion of the C++ language. Like C, Golang has pointers, structs, type inheritance, method and operator overloading. However, it is not a free form language. It requires many formatting details such as indentation and spaces. In addition, It also requires that any declared variables or imported libraries must be used. Golang starts to show its differences when it comes to types because it uses type inference.

Listing 4.1: Declaring golang variables (A Tour of Go et. al.).

```
x := 0
// or
var i int
```

Listing 4.2: Declaring C variables (A Tour of Go et. al.).

```
int x = 0
```

As we can see, the C language required the type to be specified. Whereas the Golang version required no type, but one could be specified if needed. The second way shown in listing `5.1` means that no value was assigned by the developer, yet the variable was still declared. Although the two languages are similar, Golang may look like a mixture of C and

Python. The Golang has actually been a hit amongst the Python community. It provides the speed similar to C, with the easy to read code like Python.

4.2 Conditions

Conditions are not too different from C and Python. It actually looks like a mix of the two. Like C, if statements require curly braces to define blocks of code, but it also doesn't require parentheses like Python. In addition, an if statement in Golang can declare a variable that will be out of scope once the if statement is complete.

Listing 4.3: Function with basic if statement (A Tour of Go et. al.).

```
x = -1
if x < 0 {
        fmt.Println(sqrt(-x) + "i")
}
```

Listing 4.4: Function with if statement that declares variable and loses scope when if statement is done (A Tour of Go et. al.).

```
if v := math.Pow(x, n); v < lim {
        fmt.Println(v)
}
```

Notice listing `4.4` has an if statement with a variable declaration on the same line as the condition (they are separated by semi-colons). This variable is not available outside the scope of the if statement. While it is a neat way to declare temporary variables, they will not be used in this project (but they are useful to know). Adding else if and else statements in addition to if statements is not much different. See listing 4.5 for an example.

Listing 4.5: if, else if and else statment example (A Tour of Go et. al.).

```
x := "c"
if x == "a"{
    fmt.Println(x)
} else if x == "b" {
    fmt.Println(x)
}else{
    fmt.Println(x)
}
```

The last condition to go over is the switch statement. The switch statement like other languages, is a shorter way to write a sequence of if else statements. The evaluation is from top to bottom and is similar in structure and wording to any other language that has switch statments. The additional feature that switches have in Golang is that like if statements, you can have variable declarations on the same line as the condition. See listing 4.6 for an example.

Listing 4.6: Switch statment

```
switch x := "c"; x{
case "a":
    fmt.Println(x)
case "b":
```

```
    fmt.Println(x)
default:
    fmt.Println(x)
}
```

4.3 Loops

Golang has only one loop, the for loop. Since it is a very flexible for loop, it can often look like a while loop. Usually for loops have three parts to it, a variable declaration, a condition and an iteration. Since Golangs for loops are flexible, it doesn't need any of those parts if the developer chooses to omit them. That would create an infinite for loop. The listings below show different versions of Golang's for loops.

Listing 4.7: Simple for loop (A Tour of Go et. al.).

```
sum := 0
for i := 0; i < 10; i++ {
        sum += i
}
```

Listing 4.8: For loop without variable declaration and iterator (A Tour of Go et. al.).

```
sum := 0
for ; sum < 100{
        sum += 1
}
```

Listing 4.9: Golang's version of a while loop (A Tour of Go et. al.).

```
sum := 0
for sum < 100{
        sum += 1
}
```

4.4 Functions

Like other languages, a function in Golang can take zero or more parameters. Parameters passed in must have a type declared. Instead of declaring the type after every variable, you can declare the type of the last variable in a sequence of variables if they have the same type. Golang functions can return multiple results of different types. One can also specify by name which variables are being returned from a function.

Listing 4.10: Golang function

```
func average(sum, count int)(avg int){
    avg = sum / count
    return
}
```

4.5 Pointers

Golang does have pointers like the C language. Similarly, it holds the memory address of a value. When dereferenced, it will output the value of that location in memory. Putting a & before a variable when assigning it to a pointer will assign the memory location to said pointer.

Listing 4.11: Golang pointer (A Tour of Go et. al.).

```
// int pointer declaration
var p *int
// declare int variable with value 42 and assign it to pointer
i := 42
p = &i


// sets i to 21 through the pointer p
*p = 21
```

4.6 Structs

The Golang language doesn't have any classes, but it does have structures. According to Golang's website, a structure is a collection of fields. Similar to other languages, the fields can be accessed with a dot. A huge difference between the Golang and other language structures is that Golang has a somewhat private and public feature to structures. When using a library in Golang, one would notice that all the fields have a capitalized first letter. Learning the language through the website teaches it the same way. This is because any field that doesn't start with a capitalized letter is considered a private field. When importing a library and using a structure from said library, any lowercased first letter fields are not exported to the user.

Listing 4.12: Structure declaration (A Tour of Go et. al.).

```
type Point struct {
```

```
    X,Y int  // public variables
    name int // private variable
}
```

Instead of having a field that is a pointer to a function, Golang added the feature to declare functions as part of a structure. It just requires adding a receiver to the function declaration. The naming convention for fields also applies to functions and functions for structures.

Listing 4.13: Structure function declaration (A Tour of Go et. al.).

```
func (p Point) Abs() float64 {
        return math.Sqrt(p.X*p.X + p.Y*p.Y)
}
```

4.7 Importing libraries

Importing libraries is rather unique in Golang. Rather that just specifying the name of a libary, it is preferred to specify a Github link. Unlike Node.js or Python, there isn't a file to list all the needed libraries. The Golang command line interface (also known as CLI) is required in order to install the dependencies. The CLI has an option that traverses an applications dependency graph and determines what libraries to import. While it may be a con to most developers learning the language, this has been a major reason to why Golang is so popular on Github (aside from it being an easy language to learn). When building a library, it is useful to remember it has the same feature structures do with fields. Any functions that have an capitalized first letter will be exported for use by the developer. Any functions that don't have a capitalized first letter will not be available to the developer.

4.8 Go Routines

Golang is also very famous for its concurrency support. It has a feature called go routines, which are lightweight threads managed by the GO runtime. They are much easier to write than other languages and is very important to this project for reading and updating the structures used.

Listing 4.14: Go routine (A Tour of Go et. al.).

```go
import (
	"fmt"
)
func count(x int){
	for i := 0; i < x; i++{
		fmt.Println(i)
	}
}

func main(){
  // this executes the count function twice,
  // once as a goroutine and the other as a non goroutine.
  // they are executed at the same time
  go count(1000)
  count(1000)
}
```

Chapter 5

Writing TOP in Golang (a.k.a. GTOP)

5.1 Getting Started

Like Java, Go is known for being portable. At compile time, it requires an architecture and operating system. While there may be many ways to create binaries for clients to download, this project will be require the users to compile the code. Luckily, this is very easy since one of the requirements is the operating system must be a Linux system. The other two requirements are the Go packages (for compiling the software) and an internet connection. For development and demonstration purposes, the operating system used was Ubuntu 16.10.

Listing 5.1: bash version

```bash
# Installs the golang package
sudo apt-get install golang-go

# Clones the repository and enters the folder
git clone https://github.com/Hunt4Bugs/gtop && cd gtop

# Installs dependencies for project
go get -d
```

```
# Builds binary and runs executable
go build && ./gtop
```

Use listing 5.1 above to install Go, clone the repository, install dependencies for the project, build the binary and run the executable. Notice the and condition on the last line in listing 4.1. This means if the command before it succeeds, then run the command after it. The go build portion of the last line is the way go builds the files in the current directory. If one doesn't specify what the output executable should be, it will name it the same name as the directory.

5.2 Project Structure and Libraries

Rather than writing a printing library, it is easier to find one. The command `htop` uses the `ncurses` library to display not only the table but the meters for each CPU at the top. No `ncurses` library was found for Go, but there are many other substitute libraries out there. Instead of looking for an API (application programming interface) for `ncurses`, the general goal for GTOP was to find text based user interface (TUI for short) libraries for GO. In C, the two main competitors are `termbox` and `ncurses`. For this paper, one of the most popular Go libraries on Github is `termbox-go` (the GO library for `termbox`). Many other libraries have been built on top of that. The library `termbox` has been chosen for this project, which is built on top of `termbox`. The reason for this choice is because `termbox` provides an even more minimal user interface for TUI development.

It is very easy to create a wrapper around a C function or library in Go. For this paper, the goal was to minimize or eliminate that dependency. Therefore, a library is needed to list and view the files in the process file system. Luckily, Go comes with an OS (Operating System) package that was used to list all the numbered folders and open the files in the process file system. Unfortunately, some libraries such as the OS package are limited in Go. The OS package was vital to making this project work properly, but it lacked an important feature needed to find the owner of a process.

The Go programming language has a repository on Github called `golang-standards`, which details the proper structure for various types of software projects. In this case, our project is a command line application. For command line applications, the repository recommends having a folder called `cmd` for application code. It also recommends keeping the amount of code in there to a minimum. The reasoning is there should be a folder called `pkg`, which contains reusable code for other projects. If certain libraries don't fit into a reusable category, their recommended location is a folder called `internal`.

Go has a nice way of specifying the external libraries needed. One can simply put a Github link to the needed package in the import section of the code. Once `go get` is run, it fetches all the needed libraries from Github and saves them to the $GOROOT or $GOPATH directory. Unlike Python or NodeJS, Go doesn't need a file to specify packages. For development purposes, it is much more convenient this way when working with external libraries. Unfortunately, this is also the way to import internal packages mentioned earlier. Rather than Go detecting if the package is a local folder (like Python), one should put a Github link to the internal package. Although, it has an option that goes against Go suggested standards. It still has the option to specify a local package. Instead of importing `import ("https://github.com/Hunt4Bugs/gtop/pkg/somepackage")`, a developer can use `import ("./somepackage")`. This also means the package must be in the same location of the code, thus throwing away the Golang suggested standards.

For this paper, none of these standards and options were used in the beginning when creating the tool. Instead, all files were kept inside the root folder in the repository and were split up by purpose instead. For example, functions to scan files in `/proc` were put in scan.go. This decision was initially made without knowledge of the existence of the `golang-standards` repository. In order to make this project up to `golang-standards`, the directories described above would need to be created. The `main.go` file can be put in the `/cmd/gtop` folder because that builds the executable that is needed. The library for scanning the `/proc` PID folders could be made into a public package by putting it in `/pkg` under a properly named folder for a public package. The declaration of the TermUI variables is more for internal application use. Thus, it would need to be put in the `/internal` folder. This project structure clears up the clutter that was initially created for the project. It also makes it nice to have a doc folder for documentation of the entire project. See figure 5.2 below for a proper structure for a command line application.

```
/gtop
├─ cmd
│  └─ gtop
│     └─ main.go
├─ doc
├─ internal
│  └─ tui
│     └─ layout.go
└─ pkg
   └─ proc
      └─ scan.go
         └─ deviceInfo.go
```

Figure 5.1: Proper command line application project structure.

After discovering `golang-standards`, the choice was made to make the switch and restructure the entire finished project. This would also make the simple build command a little different. Instead, a shell script was created to help install dependencies, build the project and run it. This was the finishing touch to the entire project's structure and libraries.

5.3 Hardware Information

The header of the `top` tool displays very important memory usage information about the system. In order to accomplish this in Go, the `/proc/meminfo` file needs to be read on the same one second loop delay as the process information. The file was easy to parse and understand because each line has one name and value. For the purpose of the project, the function to parse the file only grabbed the information needed. A better way to do the same operation is to grab all the information needed. It makes it reusable code for all developers to use rather than application specific. The code itself is easy and straight forward, it opens up the file and reads the contents line by line. Then a simple switch statement is used to find what type of value it is, convert it to an int type and assign it to the correct field (instead of returning values, it updates the structure).

Listing 5.2: Function to read memory info from process filesystem

```
func getMem(dev *DeviceInfo) {
  f, err := os.Open("/proc/meminfo")
  defer f.Close()
  if err == nil {
    scanner := bufio.NewScanner(f)
```

```
    for scanner.Scan() {
        text := strings.Fields(scanner.Text())
        switch text[0] {
        case "MemTotal:":
          dev.memSize, err = strconv.Atoi(text[1])
        case "MemFree:":
              dev.memFree, err = strconv.Atoi(text[1])
        case "Buffers:":
              dev.buffer, err = strconv.Atoi(text[1])
        case "Cached:":
              dev.cache, err = strconv.Atoi(text[1])
        case "SwapTotal:":
              dev.swapSize, err = strconv.Atoi(text[1])
        case "SwapFree:":
          dev.swapFree, err = strconv.Atoi(text[1])
        }
      }
      dev.swapUsed = dev.swapSize - dev.swapFree
      dev.memUsed = dev.memSize - dev.memFree
  }
}
```

5.4 PID's, Stat and Status

When the application first starts, it performs an initial scan of the process file system. This initial scan lists all the directories in the folder and begins to read them. One by one each directory is checked for the `/proc/<PID>/stat` and `/proc/<PID>/status` files. If those files exist, they are opened, parsed and the information is stored in a structure. The structures are checked to verify there is a user and string command associated with the structure. If not, the process structure that was just created is discarded instead of being stored. After the project was written, a better way was found but not implemented due to time constraints. At the moment, the user id is read from the status file, the user name is read from `/etc/passwd`. Instead, the program should have checked the owner of the process ID folder being read. Doing a `ls -l /proc` shows the owners of the files and folder to verify this is correct. This means, getting the owner id of the folder and using the OS library to get the user name (if there is such a function) would have been easier than what is currently being done.

As mentioned above, stat and status are being scanned for all processes. The code is very similar to the code shown in the Hardware Information chapter, it just reads different fields. A structure was created for the purposes of storing process information. This is used later on to display the necessary information. It also made it easier to just store a map of process ID's that is returned after the initial scan. After that, the program begins a loop of waiting one second then reading everything again. It displays the information after every iteration. A format function was written for the purposes of ordering the process by CPU usage and displaying them using string formatting. Each program has different ways of displaying formatted strings, for this application, the string `%-7s|%7s|%-7s|%-7s|%-30s` was used. This has some unfortunate side effects, it doesn't resize like `top` or `htop`. This is a feature that could be implemented in the near future. The TUI used makes it easy for the program to resize when the terminal window does. But the table layout the library provides seemed to be outdated or still have many bugs. So the decision was made to do it with string formatting which limits resizing capabilities.

5.5 Go Routine in Main function

The TUI library used at some point in time provided a way to loop on a one second interval. That unfortunately seemed to stop being supported. The library needs to call a loop function it has implemented in order to render the application and listen for user input. Thus, there was no other choice but to use a go routine to continuously update the data being used for the application. This go routine had to make its own loop to iterate on a one second interval. On every loop, it updates the data structures used, formats them into a string and calls the render function to apply the new changes to the application.

Chapter 6

Final Words

A paper by Hundt et. al. was published comparing the languages C++, Scala, Java, Go and variations of them. In that paper, they implemented the same algorithm in all four languages. It was shown that C++ won in performance by a large margin. Go was at the bottom in performance factors for all except compile time. Before discarding Go so fast, it was stated that there was extensive tuning of the C++ version. It was done at a level of sophistication above the average programmer level. Meaning it would be expensive to acquire the kind of talent and resources that Google used. In addition, there was only so much tuning done and measurements taken for Java and Scala because of their garbage collection. Go made it easier to do similar operations without so much resources and possibly time.

It was shown that coding the `top` tool in Go required minimal code and a little knowledge of the process file system. Not only did it require less resources, a new library can be written for others to use in the open source community. Unfortunately, both the `top` and `htop` tools did not accomplish such tasks. The code is application specific and hard to understand. The GTOP tool was written in a way that any beginning developer could understand and contribute. There has been a massive shift in focus to helping develop and expand the tools of the Go community. According to Stackshare et. al., not only is Go used by hobbyist developers but it is also by bigger technology companies. Companies such as Docker have written the majority of their platform and tools in Go. Widely used open source tools like Kubernetes were developed in Go as well. With the documented best practices and resources of the language, many more developers of all levels can begin to contribute to a variety of different tools.

Bibliography

[1] *A Tour of Go*. URL: https://tour.golang.org.

[2] Vivek Gite. *Understanding /etc/passwd File Format*. Aug. 2017. URL: https://www.cyberciti.biz/faq/understanding-etcpasswd-file-format/.

[3] Robert Hundt. *Loop Recognition in C++/Java/Go/Scala*.

[4] Michael Kerrisk. *PROC(5)*. Apr. 2018. URL: http://man7.org/linux/man-pages/man5/proc.5.html.

[5] Hisham H M. *How to calculate system memory usage from /proc/meminfo (like htop)*. Dec. 2016. URL: https://stackoverflow.com/questions/41224738/how-to-calculate-system-memory-usage-from-proc-meminfo-like-htop.

[6] Adam Ng. *Golang : Get all local users and print out their home directory, description and group id*. July 2017. URL: https://www.socketloop.com/tutorials/golang-get-all-local-users-and-print-out-their-home-directory-description-and-group-id.

[7] Stackshare. *Docker Tech Stack*. URL: https://stackshare.io/docker/docker.

[8] *Standard Go Project Layout*. URL: https://github.com/golang-standards/project-layout.

[9] Vangelis Tasoulas. *Accurate calculation of CPU usage given in percentage in Linux*. Apr. 2014. URL: https://stackoverflow.com/questions/23367857/accurate-calculation-of-cpu-usage-given-in-percentage-in-linux.