Originally published at https://rakyll.org/coredumps/.

Debugging is highly useful to examine the execution flow and to understand the current state of a program.

A core file is a file that contains the memory dump of a running process and its process status. It is primarily used for post-mortem debugging of a program, as well as to understand a program's state while it is still running. These two cases make debugging of core dumps a good diagnostics aid to postmortem and analyze production services.

I will use a simple hello world web server in this article, but in real life our programs might get very complicated easily. The availability of core dump analysis gives you an opportunity to resurrect a program from specific snapshot and look into cases that might only reproducible in certain conditions/environments.

Note: This flow only works on Linux at this point end-to-end, I am not quite sure about the other Unixes but it is not yet supported on macOS. Windows is not supported at this point.

Before we begin, you need to make sure that your ulimit for core dumps are at a reasonable level. It is by default 0 which means the max core file size can only be zero. I usually set it to unlimited on my development machine by typing:

```
$ ulimit -c unlimited
```

Then, make sure you have delve installed on your machine.

Here is a `main.go` that contains a simple handler and it starts an HTTP server.

```
$ cat main.go
package main

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

func main() {
    http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprint(w, "hello world\n")
    })
    log.Fatal(http.ListenAndServe("localhost:7777", nil))
}
```

Let's build this and have a binary.

```
$ go build .
```

Let's assume, in the future, there is something messy going on with this server but you are not so sure about what it might be. You might have instrumented your program in various ways but it might not be enough for getting any clue from the existing instrumentation data.

Basically, in a situation like this, it would be nice to have a snapshot of the current process, and then use that snapshot to dive into to the current state of your program with your existing debugging tools.

There are several ways to obtain a core file. You might have been already familiar with crash dumps, these are basically core dumps written to disk when a program is crashing. Go doesn't enable crash dumps by default but gives you this option on Ctrl+backslash when `GOTRACEBACK` env variable is set to "crash".

```
$ GOTRACEBACK=crash ./hello
(Ctrl+\)
```

It will crash the program with stack trace printed and core dump file will be written.

Another option is to retrieve a core dump from a running process without having to kill a process. With `gcore`, it is possible to get the core files without crashing. Let's start the server again:

```
$ ./hello &
$ gcore 546 # 546 is the PID of hello.
```

We have a dump without crashing the process. The next step is to load the core file to delve and start analyzing.

```
$ dlv core ./hello core.546
```

Alright, this is it! This is no different than the typical delve interactive. You can backtrace, list, see variables, and more. Some features will be disabled given a core dump is a snapshot and not a currently running process, but the execution flow and the program state will be entirely accessible.

```
(dlv) bt
 0  0x0000000000457774 in runtime.raise
    at /usr/lib/go/src/runtime/sys_linux_amd64.s:110
 1  0x000000000043f7fb in runtime.dieFromSignal
    at /usr/lib/go/src/runtime/signal_unix.go:323
 2  0x000000000043f9a1 in runtime.crash
    at /usr/lib/go/src/runtime/signal_unix.go:409
 3  0x000000000043e982 in runtime.sighandler
    at /usr/lib/go/src/runtime/signal_sighandler.go:129
 4  0x000000000043f2d1 in runtime.sigtrampgo
    at /usr/lib/go/src/runtime/signal_unix.go:257
 5  0x00000000004579d3 in runtime.sigtramp
    at /usr/lib/go/src/runtime/sys_linux_amd64.s:262
 6  0x00007ff68afec330 in (nil)
```

```
    at :0
 7  0x000000000040f2d6 in runtime.notetsleep
    at /usr/lib/go/src/runtime/lock_futex.go:209
 8  0x0000000000435be5 in runtime.sysmon
    at /usr/lib/go/src/runtime/proc.go:3866
 9  0x000000000042ee2e in runtime.mstart1
    at /usr/lib/go/src/runtime/proc.go:1182
10  0x000000000042ed04 in runtime.mstart
    at /usr/lib/go/src/runtime/proc.go:1152

(dlv) ls
> runtime.raise() /usr/lib/go/src/runtime/sys_linux_amd64.s:110 (PC: 0x457774)
   105:     SYSCALL
   106:     MOVL    AX, DI  // arg 1 tid
   107:     MOVL    sig+0(FP), SI   // arg 2
   108:     MOVL    $200, AX    // syscall - tkill
   109:     SYSCALL
=> 110:     RET
   111:
   112: TEXT runtime·raiseproc(SB),NOSPLIT,$0
   113:     MOVL    $39, AX // syscall - getpid
   114:     SYSCALL
   115:     MOVL    AX, DI  // arg 1 pid
```