Unix/Linux System Programming Essentials with Go

Kevser Sırça Alkış
Software Engineer
b!nalyze

DevFest 2024'
Ankara

About Me

- Software Engineer **@Binalyze**
  Previously **Hepsiburada, Wope, Digivity, Netinternet**
- Gopher
- Infrastructure & DevOps survivor

kevsersrca kev_src

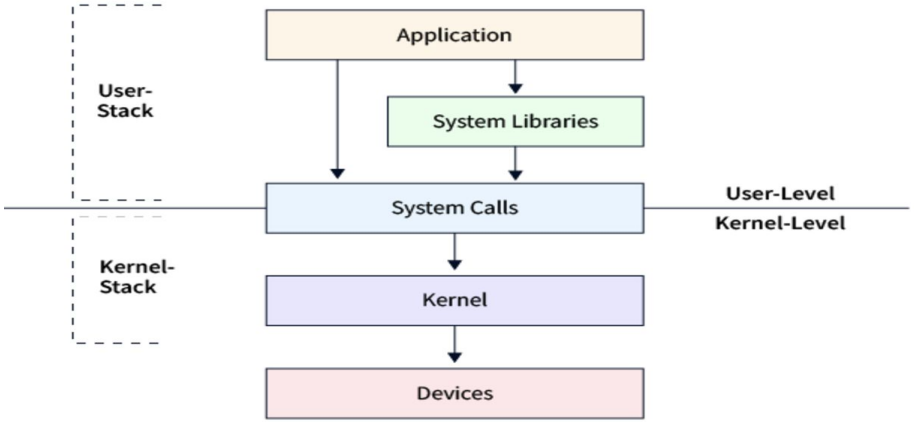
Agenda

1. Definitions
2. Path Manipulations
3. Garbage Collector
4. Processes
5. Exit Codes, Signals and Pipes

Software For Software

System programming (or systems programming) is the process of creating **computer system software**. The definition highlights two main concepts of what system applications are as follows:

- Software that is used by other software, not directly by the final user.



Protection Ring

A **protection ring** is a security architecture used in computer systems to manage access to resources by defining hierarchical privilege levels. It safeguards system resources by preventing unauthorized or faulty access.

Rings Structure:

Ring 0 (Kernel Level): The most privileged level where the operating system kernel operates, with full access to system resources.

Ring 1 and 2: Used for low-level services and device drivers.

Ring 3 (User Level): The least privileged level where user applications run, restricted from direct hardware access.

System Calls

System calls are the way operating systems provide access to the resources for the applications.
It is an **API implemented by the kernel** for accessing the hardware safely.

POSIX Standard

In order to ensure consistency between operating systems, **IEEE formalized some standards** for operating systems.

Processes

File and Directory operations

Asynchronous and synchronous I/O

Semaphores

Signal handling

Standard C

Command-line interpreter

Timers

POSIX Standards

Pipes

Shared memory

I/O port interface and control

Clocks and Timers

Threads (creation, control, cleanup, scheduling and synchronization)

Memory locking interface and Priority scheduling

Real-time signals

Windows

Windows is not natively POSIX-compliant. Efforts have been made to bridge the gap.

Open-Source Solutions

- **Cygwin**: Provides a POSIX-like environment on Windows.
- **MinGW**: Supports C applications using the Microsoft Visual C runtime.

Microsoft's Efforts

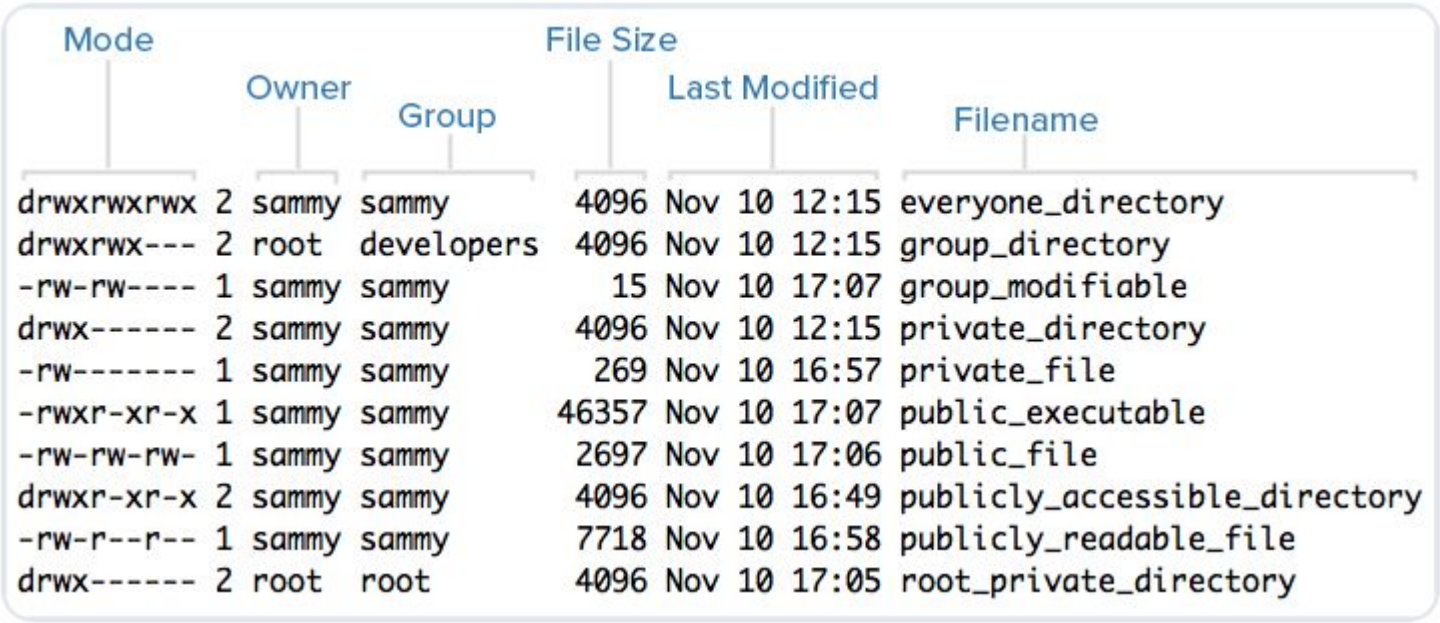
- **Microsoft POSIX Subsystem**: Early attempt for compliance.
- **Windows Subsystem for Linux (WSL)**:
  - Runs a full Linux environment on Windows.
  - Highly praised for performance and integration.

Linux and macOS

Most Linux distributions follow the **Linux Standard Base** (**LSB**), which is another standard that includes POSIX and much more, focusing on maintaining the inter-compatibility between different Linux distributions. It is not considered officially compliant because the developers didn't go into the process of certification.

However, macOS became fully compatible in 2007, and it has been POSIX-certified since then.

Resource Permissions



Understanding Mode

drwxrwxrwx

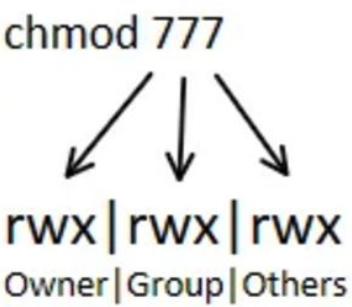
d = Directory

4 r = Read

2 w = Write

1 x = Execute

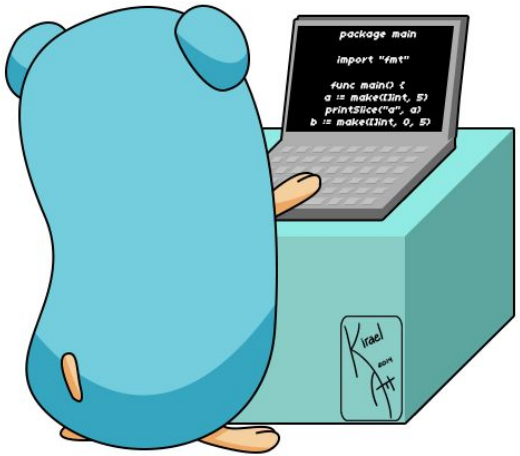
7	rwx	111
6	rw-	110
5	r-x	101
4	r--	100
3	-wx	011
2	-w-	010
1	--x	001
0	---	000



640: Owner can read and write and group can read

664: Owner and group can read and write.

Go


package main
import "fmt"
func main() {
a := make([]int, 5)
printSlice("a", a)
b := make([]int, 0, 5)
Krael

fmt

```
func main() {
    a, b := 3.0, 4.0
    h := math.Hypot(a, b)

    // Print inserts blanks between arguments when neither is a string.
    // It does not add a newline to the output, so we add one explicitly.
    fmt.Print("The vector (", a, b, ") has length ", h, ".\n")

    // Println always inserts spaces between its arguments,
    // so it cannot be used to produce the same output as Print in this case;
    // its output has extra spaces.
    // Also, Println always adds a newline to the output.
    fmt.Println("The vector (", a, b, ") has length", h, ".")

    // Printf provides complete control but is more complex to use.
    // It does not add a newline to the output, so we add one explicitly
    // at the end of the format specifier string.
    fmt.Printf("The vector (%g %g) has length %g.\n", a, b, h)

}
```

flag

```go
package main

import (
    "flag"
    "fmt"
)

func main() {
    var port int
    flag.IntVar(&port, "p", 8000, "specify port to use.  defaults to 8000.")
    flag.Parse()

    fmt.Printf("port = %d", port)
}
```

defer

Deferred function calls are pushed onto a stack. When a function returns, its deferred calls are executed in last-in-first-out order.

```
func main() {
    fmt.Println("counting")

    for i := 0; i < 10; i++ {
        defer fmt.Println(i)
    }

    fmt.Println("done")
}
```

goroutine

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    go func() {
        for i := 1; i <= 5; i++ {
            fmt.Println("Hello!")
            time.Sleep(500 * time.Millisecond)
        }
    }()

    for i := 1; i <= 5; i++ {
        fmt.Println("World!")
        time.Sleep(700 * time.Millisecond)
    }

    fmt.Println("Main function finished.")
}
```

```
World!
Hello!
World!
Hello!
World!
Hello!
World!
Hello!
World!
Main function finished.
```

run, build, install

Command	Purpose	Creates Binary	Installs Binary
`go run`	Run a program directly	No	No
`go build`	Compile a program into a binary	Yes	No
`go install`	Compile and install the binary into `$GOBIN`	Yes	Yes

build anywhere

```
GOOS=linux GOARCH=amd64 go build -o app
GOOS=windows GOARCH=386 go build -o app.exe
GOOS=darwin GOARCH=arm64 go build -o app
```

build tags

Build tags commonly used to manage platform-specific code, optional features, or dependencies.

```
//go:build linux
// +build linux

package main

import "fmt"

func main() {
    fmt.Println("Running on Linux!")
}
```

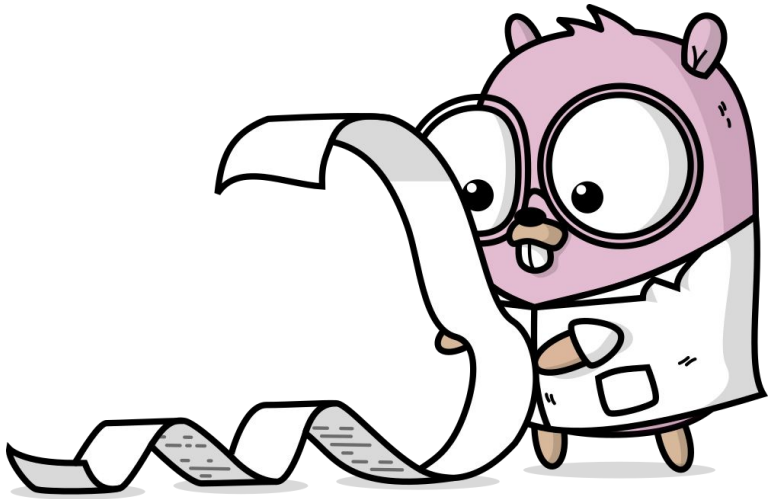
```
GOOS=linux go build -o app
```

custom build tags

```
//go:build customfeature
// +build customfeature

package main

import "fmt"

func init() {
   fmt.Println("Custom feature enabled!")
}
```

```
go build -tags=customfeature -o app .
```

Path Manipulations



working directory

Linux Command	Description	Go Equivalent	Example Code
`pwd`	Print the current directory	`os.Getwd()`	`wd, err := os.Getwd()`

```go
wd, err := os.Getwd()
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("working dir:", wd)
```

create directory

Linux Command	Description	Go Equivalent	Example Code
`mkdir -p <dir>`	Create nested directories	`os.MkdirAll(name string, perm os.FileMode)`	`err = os.MkdirAll("parent/child", 0o755)`
`mkdir <dir>`	Create a directory	`os.Mkdir(name string, perm os.FileMode)`	`err =os.Mkdir("example.txt", 0o755)`

rename and move

Linux Command	Description	Go Equivalent	Example Code
`mv <src> <dst>`	Move or rename a file or directory	`os.Rename(oldpath, newpath string)`	`err = os.Rename("old.txt", "new.txt")`

```go
package main

import "os"

func main() {
    err := os.Rename("kev.txt", "/tmp/output-20241215.txt")
    if err != nil {
        panic(err)
    }
}
```

path/filepath vs path package

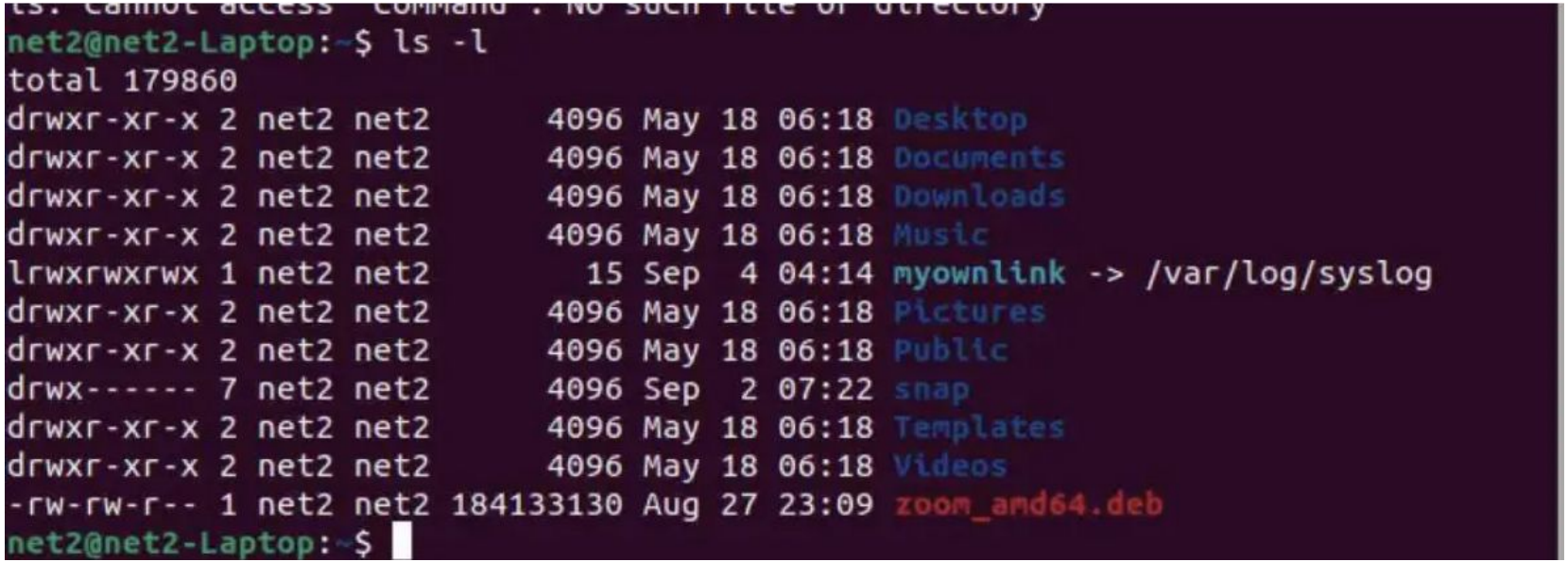
path/filepath

- Handles **local filesystem paths**.
- **Platform-Dependent**: Respects OS-specific separators (\ on Windows, / on Linux).
- Filesystem operations.

path

- Handles **generic or URL paths**.
- **Platform-Independent**: Always uses / as the separator.
- Web paths or cross-platform path manipulation

absolute path and symlinks



absolute path and symlinks

```
func main() {
    path := "go1.21.13"
    err := os.Symlink("/usr/local/go/bin/go", path)
    if err != nil {
        fmt.Println("Error:", err)
    }

    if realPath, err := filepath.EvalSymlinks(path); err != nil {
        path = realPath
    }
    fmt.Println("Real Path:", path)
}
```

absolute path and symlinks

```go
package main

import (
    "fmt"
    "path/filepath"
)

func main() {
    if !filepath.IsAbs("test.sh") {
        if abs, err := filepath.Abs("test.sh"); err == nil {
            fmt.Println(abs)
        }
    }
}
```

```
/Users/kevsersirca/go/src/_tmp/test.sh
```

file create,write and close

```go
file, err := os.Create("output.txt")
if err != nil {
    fmt.Println("Error creating file:", err)
    return
}
defer file.Close()

data := "Hello, World!"
n, err := file.Write([]byte(data))
if err != nil {
    fmt.Println("Error writing to file:", err)
    return
}
fmt.Printf("Wrote %d bytes\n", n)
```

Garbage Collector

Garbage Collector

Memory Management

The operating system handles the **primary** and **secondary** memory usage of the applications. It keeps track of how much of the memory is used, by which process, and what parts are free. It also handles allocation of new memory from the processes and memory de-allocation when the processes are complete.

- Single allocation
- Partitioned allocation
- Paged memory

Go supports Garbage Collection (GC) so you do not have to deal with memory allocation and deallocation.

Virtual Memory

Unix uses the paged memory management technique, abstracting its memory for each application into contiguous virtual memory. It also uses a technique called swapping, which extends the virtual memory to the secondary memory using a swap file

The top command shows details about available memory, swap, and memory consumption for each process:

- RES is the physical primary memory used by the process.
- VIRT is the total memory used by the process, including the swapped memory, so it's equal to or bigger than RES.
- SHR is the part of VIRT that is actually shareable

GC

GC (Garbage Collection) is an automatic memory management mechanism used in programming languages. It identifies memory that is no longer needed (unreachable or unreferenced objects) and reclaims it for reuse.

https://tip.golang.org/doc/gc-guide

Marking a Variable for GC

```go
package main

func example() {
    data := make([]byte, 1024)
}

// data becomes unreachable after the function ends and
can be collected.
```

Marking a Variable for GC

```
package main

func main() {
    var data = make([]byte, 1<<20) // Allocate 1 MB
    …

    data = nil
    // The data is now eligible for garbage collection
}
```

Trigger the GC Manually

```go
package main

import "runtime"

func main() {
    var data = make([]byte, 1<<20) // Allocate 1 MB
    data = nil                     // Remove reference

    runtime.GC()                   // Force garbage collection
}
```

https://tip.golang.org/doc/gc-guide

runtime.MemStats

```
func logMemoryUsage() {
    var memStats runtime.MemStats
    runtime.ReadMemStats(&memStats)
    fmt.Printf("Time: %v | Alloc = %v MiB | Sys = %v MiB | NumGC = %v\n",
        time.Now().Format("15:04:05"),
        memStats.Alloc/1024/1024, // Current allocation (MiB)
        memStats.Sys/1024/1024,   // Total allocation (MiB)
        memStats.NumGC)           // Total count of GC runs
}
```

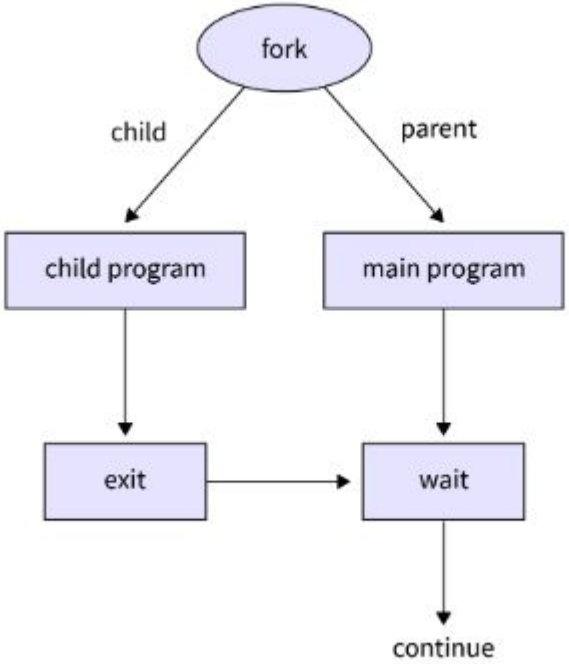
Processes

Processes

When an application is launched, it becomes a process: a special instance provided by the operating system that includes all the resources that are used by the running application.

```
UID          PID  PPID  C STIME TTY          TIME CMD
root           1     0  0 10:00 ?        00:00:01 /sbin/init
user        1234     1  0 10:05 ?        00:00:10 go run main.go
user        5678  1234  0 10:10 pts/1    00:00:03 top
```

Processes Life Cycle



processes

```
func main() {
    fmt.Println("Current PID:", os.Getpid())
    fmt.Println("Current Parent PID:", os.Getppid())
    fmt.Println("User ID:", os.Getuid())
    fmt.Println("Group ID:", os.Getgid())
}
```

child process

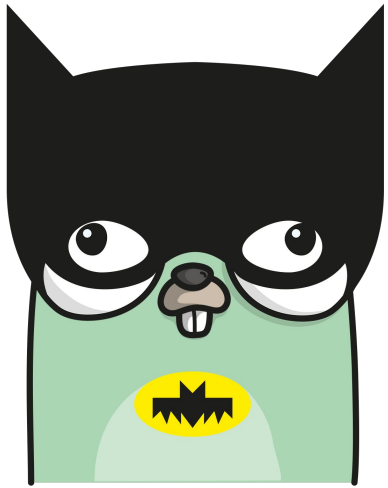
```go
import (
    "fmt"
    "os/exec"
)

func main() {
    cmd := exec.Command("ls", "-l")
    if err := cmd.Start(); err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Cmd: ", cmd.Args[0])
    fmt.Println("Args:", cmd.Args[1:])
    fmt.Println("PID: ", cmd.Process.Pid)
    cmd.Wait()
}
```

stdin and stdout

Standard input can be used to send some data from the application to the child process.
Standard output represents destination sends its output data.

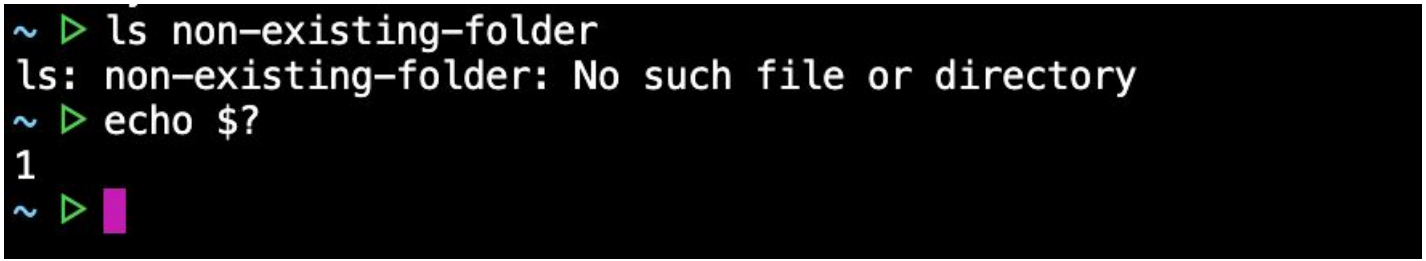
```
b := bytes.NewBuffer(nil)

cmd := exec.Command("cat")
cmd.Stdin = b
cmd.Stdout = os.Stdout
fmt.Fprintf(b, "Hello World! I'm using this memory address: %p", b)
if err := cmd.Start(); err != nil {
  fmt.Println(err)
  return
}
cmd.Wait()
```

Exit Codes, Signals and Pipes



sending exit codes

Applications communicate their result to the operating system by returning a value called **exit status**. This is an integer value passed to the parent process when the process ends.
0: Indicates that the process completed successfully.
Non-zero: Indicates an error or abnormal termination. Different non-zero values often correspond to specific error types.



sending exit codes

Exit codes are the way in which a process notifies its parent about its status after terminating.

```
import (
    "fmt"
     "os"
)

func main() {
    fmt.Println("Hello world!")
    os.Exit(1)
}
```

sending exit codes

```
func main() {
    fmt.Println("Hello, playground")
    os.Exit(-1)
}
```

This will have an exit status of 255 even if the function argument is -1 because (-1)%256=255. This happens because the exit code is an 8-bit number (0, 255).

panic

stack unwinding

```
func main() {
    defer fmt.Println("Hello, playground")
    panic("panic")
}
```

reading child process exit code

```
func main() {
    cmd := exec.Command("ls", "nonexistent-file")
    err := cmd.Run()

    if err != nil {
        if exitError, ok := err.(*exec.ExitError); ok {
            exitCode := exitError.ExitCode()
            fmt.Printf("Child process exited with code: %d\n", exitCode)
        }
    }
}
```

signals

Exit codes connect processes and their parents, but signals make it possible to interface any process with another, including itself. They are also asynchronous and unidirectional, but they represent communication from the outside of a process.

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGEMT       8) SIGFPE       9) SIGKILL     10) SIGBUS
11) SIGSEGV     12) SIGSYS      13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGURG      17) SIGSTOP     18) SIGTSTP     19) SIGCONT     20) SIGCHLD
21) SIGTTIN     22) SIGTTOU     23) SIGIO       24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGPWR      30) SIGUSR1
31) SIGUSR2     32) SIGRTMIN    33) SIGRTMIN+1  34) SIGRTMIN+2  35) SIGRTMIN+3
36) SIGRTMIN+4  37) SIGRTMIN+5  38) SIGRTMIN+6  39) SIGRTMIN+7  40) SIGRTMIN+8
41) SIGRTMIN+9  42) SIGRTMIN+10 43) SIGRTMIN+11 44) SIGRTMIN+12 45) SIGRTMIN+13
46) SIGRTMIN+14 47) SIGRTMIN+15 48) SIGRTMIN+16 49) SIGRTMAX-15 50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4
61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1  64) SIGRTMAX
```

signals

```
signalChan := make(chan os.Signal, 1)

signal.Notify(signalChan, os.Interrupt, syscall.SIGTERM)

go func() {
    sig := <-signalChan
    fmt.Printf("Sinyal catch: %s\n", sig)

    fmt.Println("closing application")
    os.Exit(0)
}()
```

signals

```go
p, err := os.FindProcess(pid)
if err != nil {
    panic(err)
}
if err = p.Signal(syscall.SIGTERM); err != nil {
    panic(err)
}
```

```go
func os.FindProcess(pid int) (*os.Process, error)
```

FindProcess looks for a running process by its pid.

The Process it returns can be used to obtain information about the underlying operating system process.

On Unix systems, FindProcess always succeeds and returns a Process for the given pid, regardless of whether the process exists. To test whether the process actually exists, see whether p.Signal(syscall.Signal(0)) reports an error.

`os.FindProcess` on pkg.go.dev

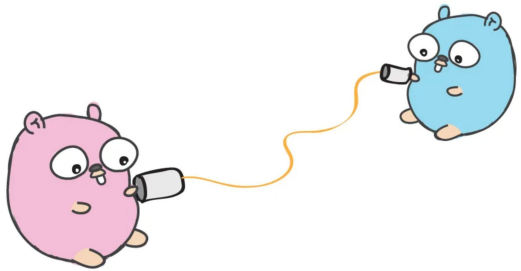
pipes

Pipes are the last unidirectional communication method between processes. As the name suggests, pipes connect two ends – a process input with another process output – making it possible to process on the same host to communicate in order to exchange data.

```
cat input.txt | sort | wc -l
```

pipes

```
reader, writer := io.Pipe()
```



pipes

```
reader, writer := io.Pipe()

go func() {
    defer writer.Close()
    writer.Write([]byte("Hello, Pipe!"))
}()

buf := make([]byte, 20)
n, err := reader.Read(buf)
if err != nil {
    fmt.Println("Error reading from pipe:", err)
    return
}

fmt.Println("Read from pipe:", string(buf[:n]))
```

Thanks!

Kevser Sırça Alkış

@kevsrca

@kev_src