Systems Programming

Concepts of Programming Languages

30 November 2020

Bernhard Saumweber
Rosenheim Technical University

What is Systems Programming?

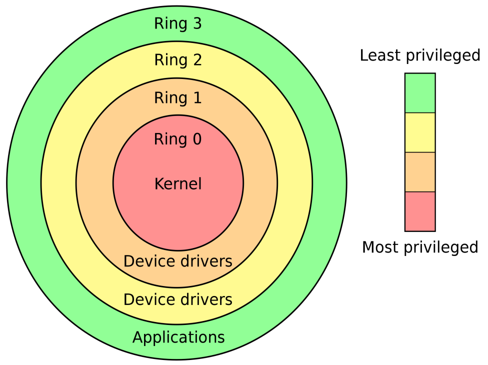
Definition

"Systems programming involves designing and writing computer programs that allow the computer hardware to interface with the programmer and the user,
leading to the effective execution of application software on the computer system.

Typical programs include the operating system and firmware, programming tools such as compilers, assemblers, I/O routines, interpreters, scheduler, loaders and linkers as well as the runtime libraries of the computer programming languages."

Systems Programming Today

- Todays OSs are built on a kernel of Assembler and C-Code (Linux, OSX/Darwin, Win32)
- This includes the Kernel, Drivers, Graphics Subsystem, HAL ...
- The higher layers are built in languages with good support to call C or Assembler code (C, C++, Objective C)
- Tools and Low Level Applications on top are typically written in languages like Python, Perl, C++, Go
- Highlevel Applications are written in any Programming Language like Java, Ruby, JS ...



Calling C Code from Go with cgo

Calling C Code from Go with cgo

Import the pseudo package C in your Go source and include the corresponding C header files (comment) to call a C function.

```go
package rand

// #include <stdlib.h>
import "C"
func Random() int {
    var r C.long = C.random() // calls "int rand(void)" from the C std library
    return int(r)
}

func Seed(i int) {
    C.srandom(C.uint(i))
}
```

To build cgo packages, just use go build or go install as usual. The go tool recognizes the special "C" import and automatically uses cgo for those files.

Calling C Code from Go with cgo

```go
package main

// #include &lt;stdlib.h&gt;
import "C"
import "fmt"

func Seed(i int) {
    C.srandom(C.uint(i))
}

func Random() int {
    var r C.long = C.random() // calls "int rand(void)" from the C std library
    return int(r)
}

func main() {
    Seed(1)
    fmt.Printf("random int from C is %d", Random())
}
```

Run

Controlling Linking with Build Flags

Build flags for the cgo command can be set in the Go source file as comments:

```go
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo amd64 386 CFLAGS: -DX86=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

When calling functions from a library outside libc, the Go programm must be linked with that library (here: png.lib/png.so).

This can be done by adding cgo build flags to the comment section.

Mapping the C namespace to Go

- Everything declared in the C code is available in the C pseudo-package
- Fundamental C data types have their counterpart, e.g. int → C.int, unsigned short → C.ushort, etc.
- The Go equivalent to void * is unsafe.Pointer
- typedefs are available under their own name
- structs are available with a struct_ prefix, e.g. struct foo → C.struct_foo, same goes for unions and enums

Conversion between C and Go strings

The C package contains conversion functions to convert Go to C strings and vice versa

Also: opaque data (behind void *) to []byte

```
// Go string to C string; result must be freed with C.free()
func C.CString(string) *C.char

// C string to Go string
func C.GoString(*C.char) string

// C string, length to Go string
func C.GoStringN(*C.char, C.int) string

// C pointer, length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

golang.org/cmd/cgo/ (https://golang.org/cmd/cgo/)

Platform/OS Programming with Go

Platform independent programming with Go

The packages os and runtime encapsulate platform dependent APIs

```go
// Copyright 2018 Johannes Weigend
// Licensed under the Apache License, Version 2.0
package main

import (
    "fmt"
    "os"
)

func main() {
    pid := os.Getpid()
    fmt.Println("process id: ", pid)
}
```

Run

The OS Package

Access to the standard I/O streams

```
os.Stdin
os.Stdout
os.Stderr
```

Functions for executing processes (os/exec)

```
cmd := exec.Command(path, args)
```

Access to the environment

```
os.Getenv("FOO")
os.Setenv("FOO", "1")
```

Access to files, directories, pipes, ...

```
f, err := os.Open("/tmp/dat") // f.Read() / f.Write() / f.Seek()
```

golang.org/pkg/os/ (https://golang.org/pkg/os/)

The Runtime Package

Package runtime contains operations that interact with Go's runtime system, such as functions to control goroutines. It also includes the low-level type information used by the reflect package

golang.org/pkg/runtime/ (https://golang.org/pkg/runtime/)

Platform dependent programming with Go

- The Go syscall package has platform dependent functionality which is different per OS
- Go has a wrapper package to use the C syscall function (#include<sys/syscall>)

```go
// +build linux

// Copyright 2018 Johannes Weigend, Johannes  Siedersleben
// Licensed under the Apache License, Version 2.0
package main

import (
    "fmt"

    "golang.org/x/sys/unix"
)

func main() {
    pid, _, _ := unix.Syscall(39, 0, 0, 0)
    fmt.Println("process id: ", pid)
}
```

Run

golang.org/pkg/syscall/ (https://golang.org/pkg/syscall/)

godoc.org/golang.org/x/sys (https://godoc.org/golang.org/x/sys)

Building a Container from the Scratch

What is a Container?

What is a Container?

- Containers let you isolate an application so that it’s under the impression it’s running on its own private machine. In that sense, a container is similar to a virtual machine (VM), but it uses the operating system kernel on the host rather than having its own.
- Containers are started from container images. An image packs the application together with the required libraries and tools.
- In many cases a full unix filesystem is packed inside the image.
- An image for a Go application could be empty, since Go has static linking.
- The required dependend code is linked into the binary.

Change Root (chroot)

A chroot on Unix operating systems is an operation that changes the apparent root directory for the current running process and its children. A program that is run in such a modified environment cannot name (and therefore normally cannot access) files outside the designated directory tree. The term "chroot" may refer to the chroot(2) system call or the chroot(8) wrapper program. The modified environment is called a chroot jail.\

```
# chroot new_root [command]
# mkdir -p /home/user/jail
# chroot /home/user/jail /bin/bash
  => chroot: failed to run command ‘/bin/bash’: No such file or directory
```

Change Root can not start because there is no /bin/bash inside the new root!
The solution is to copy bash inclusive dependent libraries inside the directory jail tree

```
$ mkdir /home/user/jail/bin
$ cp /bin/bash /home/user/jail/bin
```

Many Applications need Dependent Libraries to Run

List dependent libraries for /bin/bash

```
ldd /bin/bash
```

```
linux-vdso.so.1 =>  (0x00007fff11bff000)
libtinfo.so.5 => /lib64/libtinfo.so.5 (0x0000003728800000)
libdl.so.2 => /lib64/libdl.so.2 (0x0000003d56400000)
libc.so.6 => /lib64/libc.so.6 (0x0000003d56800000)
/lib64/ld-linux-x86-64.so.2 (0x0000003d56000000)
```

Copy libraries to jail root

```
mkdir /home/user/jail/lib64
$ cp /lib64/{libtinfo.so.5,libdl.so.2,libc.so.6,ld-linux-x86-64.so.2} /home/user/jail/lib64
```

Start /bin/bash with new root

```
# chroot /home/user/jail
# pwd => /
# ls => /bin, /lib64
```

The bash shell can only access the jail file system!

CGroups

- Control cgroups, usually referred to as cgroups, are a Linux kernel feature which allow processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs.
- Grouping is implemented in the core cgroup kernel code, while resource tracking and limits are implemented in a set of per-resource-type subsystems (Memory, CPU, and so on).

CGroups Features

Resource limiting
groups can be set to not exceed a configured memory limit, which also includes the file system cache

Prioritization
some groups may get a larger share of CPU utilization or disk I/O throughput

Accounting
measures a group's resource usage, which may be used, for example, for billing purposes

Control
freezing groups of processes, their checkpointing and restarting

Linux Namespaces

Namespaces are a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources. The feature works by having the same name space for these resources in the various sets of processes, but those names referring to distinct resources. Examples of resource names that can exist in multiple spaces, so that the named resources are partitioned, are process IDs, hostnames, user IDs, file names, and some names associated with network access, and interprocess communication.

How to build a Container in Go

The code shows the main functionality of the "docker run" command:

```go
// go run main.go run <cmd> <args>
func main() {
    switch os.Args[1] {
    case "run":
        run()
    case "child":
        child()
    default:
        panic("help")
    }
}
```

How to build a Container in Go

The run() function starts a new process (a clone) with a unique namespace
The child() function function is called in the new process and configures the new namespace (setHostname ...)

```go
func run() {
    log.Printf("Running %v \n", os.Args[1:])

    cmd := exec.Command("/proc/self/exe", append([]string{"child"}, os.Args[2:]...)...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
    cmd.SysProcAttr = &syscall.SysProcAttr{
        // Does not compile on OSX
        Cloneflags:   syscall.CLONE_NEWUTS | syscall.CLONE_NEWPID | syscall.CLONE_NEWNS,
        Unshareflags: syscall.CLONE_NEWNS,
    }

    must(cmd.Run())
}
```

Clone flags

- CLONE_NEWNS (since Linux 2.4.19): If CLONE_NEWNS is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent. If CLONE_NEWNS is not set, the child lives in the same mount namespace as the parent.
- CLONE_NEWUTS (since Linux 2.6.19): If CLONE_NEWUTS is set, then create the process in a new UTS namespace, whose identifiers are initialized by duplicating the identifiers from the UTS namespace of the calling process. If this flag is not set, then (as with fork(2)) the process is created in the same UTS namespace as the calling process. This flag is intended for the implementation of containers.
- CLONE_NEWPID (since Linux 2.6.24): If CLONE_NEWPID is set, then create the process in a new PID namespace. If this flag is not set, then (as with fork(2)) the process is created in the same PID namespace as the calling process. This flag is intended for the implementation of containers.

The Child Process Initializer

```go
func child() {
    log.Printf("Running %v \n", os.Args[1:])

    cg() // Initialize CGroup limits

    cmd := exec.Command(os.Args[2], os.Args[3:]...)
    cmd.Stdin = os.Stdin
    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr

    // We are in an independent clone -> Setting hostname is safe!
    must(syscall.Sethostname([]byte("container")))
    must(syscall.Chroot("/opt/alpinefs")) // container local file system (alpine linux)

    must(os.Chdir("/"))
    must(syscall.Mount("proc", "proc", "proc", 0, "")) // support for the ps command
    must(cmd.Run())

    must(syscall.Unmount("proc", 0))
}
```

CGroup Initialization limits Resource Consumption of Container

```go
func cg() {
    cgroups := "/sys/fs/cgroup/"
    pids := filepath.Join(cgroups, "pids")
    os.Mkdir(filepath.Join(pids, "container"), 0755)
    must(ioutil.WriteFile(filepath.Join(pids, "container/pids.max"), []byte("20"), 0700))
    // Removes the new cgroup in place after the container exits
    must(ioutil.WriteFile(filepath.Join(pids, "container/notify_on_release"), []byte("1"), 0700))
    must(ioutil.WriteFile(filepath.Join(pids, "container/cgroup.procs"), []byte(strconv.Itoa(os.Getpid()
    ...
}
```

See also the Video of Liz Rice

www.youtube.com/watch?v=Utf-A4rODH8 (https://www.youtube.com/watch?v=Utf-A4rODH8)

Summary

- Go is an excellent language for system programming
- The cgo C interface is easy to use and does not require separate tooling
- You can implement the main functionality of a Linux container in less than 50 lines
- The os, runtime packages enable platform independent programming (like in Java)
- The syscall and x/sys/. packages enable platform dependent programming (to access platform specific functionality)

Thank you

Bernhard Saumweber
Rosenheim Technical University
bernhard.saumweber@qaware.de (mailto:bernhard.saumweber@qaware.de)
http://www.qaware.de (http://www.qaware.de)