

Manual Técnico

Práctica 1

Benaventi Bernal Fuentes Roldan 201021212

Juan Pablo Rojas Chinchilla 201900289

Luis Daniel Ernesto Castellanos Galindo 201902238

Lourdes Rosario Velásquez Melini 201906564

Go

Es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C, que intenta ser dinámico como Python y con el rendimiento de C o C++. Ha sido desarrollado por Google y sus diseñadores iniciales fueron Robert Griesemer, Rob Pike y Ken Thompson.



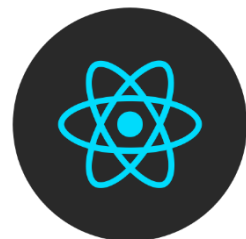
C



El lenguaje de programación C fue creado por Brian Kernighan y Dennis Ritchie a mediados de los años 70. El lenguaje C es un lenguaje para programadores en el sentido de que proporciona una gran flexibilidad de programación y una muy baja comprobación de incorrecciones, de forma que el lenguaje deja bajo la responsabilidad del programador acciones que otros lenguajes realizan por sí mismos.

React

React te ayuda a crear interfaces de usuario interactivas de forma sencilla. Diseña vistas simples para cada estado en tu aplicación, y React se encargará de actualizar y renderizar de manera eficiente los componentes correctos cuando los datos cambien.

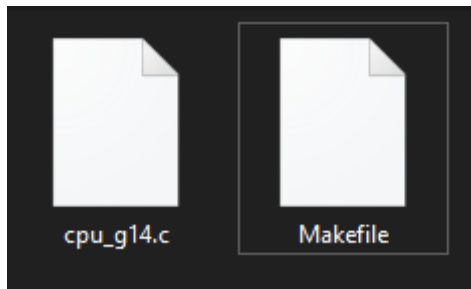


Módulos


Estos funcionan debido a la generación de archivos dentro del directorio /proc de Linux el cual tiene un peso de 0 bytes de igual forma contiene información. El directorio /proc contiene información de los procesos y aplicaciones que se están ejecutando en un momento determinado en el sistema, en realidad lo que almacena son archivos virtuales es por lo que su peso es de 0 bytes

Para la creación de los módulos se necesita tener el compilador de C y los headers necesarios para la obtención de la información, para poder trabajar esto se debe tener una estructura para poder iniciar el módulo.

- Antes que nada, se deberá tener 2 archivos los cuales son:
 - Un archivo .c el cual es que contiene el módulo
 - Un archivo Makefile el cual contiene la información para la generación del módulo por comando.



- Makefile:
Para que funcione el archivo se necesita que tenga esta estructura en específico, para que añada el módulo y elimine el módulo.

```
> cpu >  Makefile
1  obj-m += cpu_g14.o
2
3  all:
4      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
5  clean:
6      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

- Archivo en .c:
Normalmente este archivo al inicio tiene las importaciones de las librerías que se desean ya que estas importaciones podemos obtener los struct con información del sistema y también para la generación de los archivos. Además de otras funciones que tienen según lo programado.

```
#include <linux/init.h>
#include <linux/sched/signal.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/mm.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/version.h>
#include <linux/fs.h>
```

Cuando se monta un módulo aparece un mensaje, para que se llegue a mostrar el mensaje se necesita crear un objeto con la estructura `proc_ops`, luego de esto se realiza una función la cual se le pone lo que se quiera mostrar. Así que cuando se inicie el módulo se imprimirá.

Al desmontar un módulo será lo mismo solamente cambia una cosa y es que cuando se desmonte se necesita la función que imprima el desmontar.

Módulo RAM

Se necesita utilizar la estructura de datos de `sysinfo`, además se necesita la RAM total y la RAM libre. Aquí se necesita poner una estructura la cual funciona para imprimir en el archivo y obtener la información aquí se pone los datos:

- `MODULE_LICENSE("GPL");`
- `MODULE_AUTHOR("Grupo 14");`
- `MODULE_DESCRIPTION("modulo de RAM");`
- `Struct sysinfo info`

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Grupo 14");
MODULE_DESCRIPTION("modulo de RAM");
```

Luego tenemos una función que es llamada `proc_ram_data` la cual se ejecutara cada vez que se lea el archivo con el comando `CAT`. En esta función le vamos a dar un formato `JSON` para que se lea con dicho comando. Para que este archivo se imprima se usa otra función la cual manda a imprimir el archivo, sería la que terminaría el funcionamiento del módulo.

Módulo CPU

Para la realización de este módulo, necesitamos una gran parte de información, de igual manera aquí se necesitarán importaciones que nos ayudarán con la funcionalidad. Aquí usamos la estructura de task_struct y list_head. Para recorrer estas estructuras usamos diversas funciones nativas las cuales son las que importamos como:

- linux/sched.h
- linux/mm.h

```
cpu > C cpu_g14.c
1  #include <linux/init.h>
2  #include <linux/sched/signal.h>
3  #include <linux/proc_fs.h>
4  #include <linux/seq_file.h>
5  #include <linux/mm.h>
6  #include <linux/module.h>
7  #include <linux/kernel.h>
8  #include <linux/sched.h>
9  #include <linux/version.h>
10 #include <linux/fs.h>
```

Usamos un objeto en el modelo del CPU para dar formato de JSON. Usamos un ciclo para recorrer la lista de procesos que nos dan para que así se recorra a los hijos. Para obtener la información de los hijos usamos un list_for_each.

Montar los módulos

- make
Como primer paso se debe ejecutar el comando make tanto en la ram como en la cpu. Esto generara los archivos necesarios para montar el módulo.
- insmod
Luego se ejecutará el comando insmod es el cual insertará el módulo, se debe cargar el archivo .ko
- dmesg
También podemos comprobar que se está montando el mensaje
- rmmod
Se puede comprobar que está imprimiendo el mensaje de salida si ejecutamos el comando rmmod y luego comprobamos con dmesg.
- cat
Al hacer cat en el directorio /proc se puede comprobar que el módulo esta funcionando

Go

- `ps -eo pcpu | sort -k 1 -r | head -50`

Sirve para el uso del CPU

- `free -m | head -n2 | tail -1 | awk '{print $6}'`

Sirve para la memoria cache

- `id -nu {usuario}`

Sirve para identificar el usuario

- `strace`

Para utilizar el comando `strace` es necesario la utilizacion de estos métodos, estos luego son llamados en nuestro api, por medio de un web socket, hacia nuestra aplicación en react, esta api fue levantada sobre gorilla/mux al igual que los web socket.

```
func (s SyscallCounter) Init() SyscallCounter {
    s = make(SyscallCounter, maxSyscalls)
    return s
}

func (s SyscallCounter) Inc(syscallID uint64) error {
    if syscallID > maxSyscalls {
        return fmt.Errorf("invalid syscall ID (%x)", syscallID)
    }

    s[syscallID]++
    return nil
}

func (s SyscallCounter) Print() []Strace {
    var list []Strace
    w := tabwriter.NewWriter(os.Stdout, 0, 0, 8, ' ', tabwriter.AlignRight|tabwriter.Debug)
    for k, v := range s {
        if v > 0 {
            name, _ := seccomp.ScmpSyscall(k).GetName()
            fmt.Fprintf(w, "%d\t%s\n", v, name)
            var strace Strace
            strace.Name = name
            strace.Recurrence = v
            list = append(list, strace)
        }
    }
    w.Flush()
    return list
}

func (s SyscallCounter) GetName(syscallID uint64) string {
    name, _ := seccomp.ScmpSyscall(syscallID).GetName()
    return name
}
```