MANUAL TECNICO

G10

Pracica1

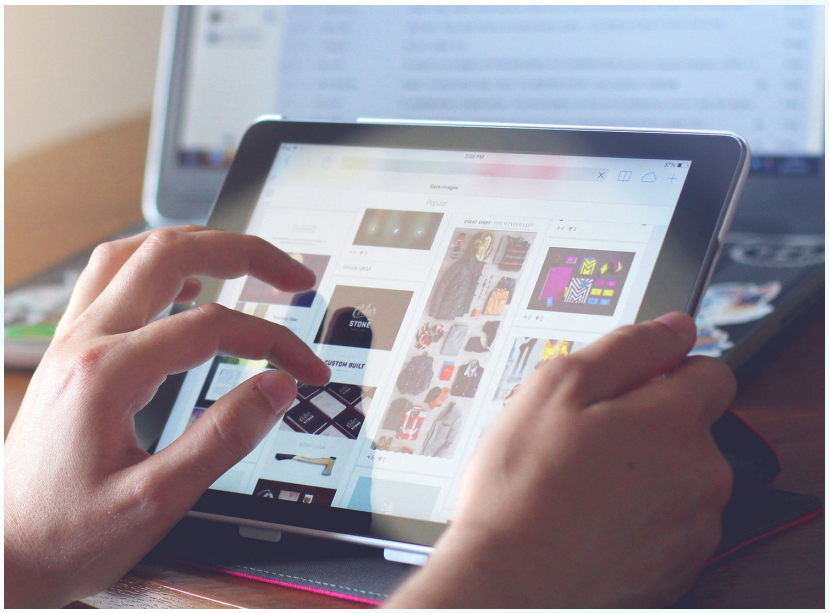
Módulos

Un módulo del kernel es un fragmento de código o binarios que pueden ser cargado y eliminados del kernel según las necesidades de este. Tienen el objetivo de extender sus funcionalidades son fragmentos de código que pueden ser cargados y eliminados del núcleo bajo demanda. Extienden la funcionalidad del núcleo sin necesidad de reiniciar el sistema. Esto es gracias a que el kernel tiene un diseño modular, cuando se instala un nuevo componente o se inicia la computadora los módulos son cargados de forma dinámica para que funcionen de forma transparente.

RAM

```
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/mm.h>

#define  BUFSIZE 150
```

Esta primera parte del código muestra las librerías utilizadas en todo el modulo las cuales permetiran leer la información del sistema



```
MODULE_LICENSE("A2");
MODULE_AUTHOR("MIGUEL ANGEL");
MODULE_DESCRIPTION("Modulo que muestra la informacion de la ram");
MODULE_VERSION("1.0");
```

Sea agrega la información del modulo que describe información detalalda

```
struct sysinfo inf;
static int write_ram(struct seq_file * archivo, void *v){
    si_meminfo(&inf);
    long total_memoria = (inf.totalram * 4);
    long memoria_libre = (inf.freeram * 4);
    seq_printf(archivo, "{\n\t\"total\":%8lu,\n", total_memoria/1024);
    seq_printf(archivo, "\t\"uso\":%8lu\n}", (total_memoria-memoria_libre)/1024);
    return 0;
}
```

Esta función realiza la lectura de información de la memoria ram usando una librería que por medio de una estructura definida obtiene los datos y posteriormente guarda la información en un archivo que será utilizado después para su lectura

```
static int my_proc_open(struct inode *inode, struct file*file){
    return single_open(file, write_ram, NULL);
}

static ssize_t my_proc_write(struct file *file, const char __user *buffer, size_t count, loff_t *f_pos){
    return 0;
}

static struct file_operations my_fops = {
    .owner = THIS_MODULE,
    .open = my_proc_open,
    .release = single_release,
    .read = seq_read,
    .llseek = seq_lseek,
    .write = my_proc_write
};
```

Estas funciones permiten la creación del archivo que tendrá la información de la ram que se leyeron anteriormente

```
static int ram_mod_init(void){
    proc_create("ram_p1", 0, NULL, &my_fops);
    printk(KERN_INFO "ram_p1: Hola mundo, somos el grupo 10");
    return 0;
}

static void ram_mod_exit(void){
    remove_proc_entry("ram_p1",NULL);
    printk(KERN_INFO "ram_p1: Sayonara mundo, somos el grupo 10");
}


module_init(ram_mod_init);
module_exit(ram_mod_exit);
```

Se crean las funciones principales y sus llamadas para la creación del modulo la cual tiene una función de inicio y de final con un respectivo mensaje

CPU

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

Esta primera parte del código muestra las librerías utilizadas en todo el modulo las cuales permetiran leer la información del sistema

```c
MODULE_LICENSE("P1");
MODULE_AUTHOR("MIGUEL ANGEL");
MODULE_DESCRIPTION("Modulo que muestra la los procesos");
MODULE_VERSION("1");
```

Sea agrega la información del modulo que describe información detalalda

```c
struct task_struct *task; //Estructura definida en sched.h para tareas/procesos
struct task_struct *childtask; //Estructura necesaria para iterar a travez de procesos secundarios
struct task_struct *memtask;
struct list_head *list; //Estructura necesaria para recorrer cada lista de tareas tarea->estructura de hijos
struct sysinfo inf;
```

Se inicializan las estructuras que se utilizaran para la lectura de procesos y sus hijos, al igual que información de memoria que los mismos ocupan

```c
static int write_cpu(struct seq_file * cpufile, void *v){
    unsigned long rss;
    si_meminfo(&inf);

    int p = 0;

    int tot = 0;
    int eje = 0;
    int sus = 0;
    int det = 0;
    int zom = 0;

    seq_printf(cpufile, "{\n\"procesos\":[\n");
    for_each_process( task ){···

    seq_printf(cpufile, "}\n");
    seq_printf(cpufile, "],\n");
    seq_printf(cpufile, "\"datos\":{\n");
    seq_printf(cpufile, "\"total\": %i,\n", tot);
    seq_printf(cpufile, "\"ejecucion\": %i,\n", eje);
    seq_printf(cpufile, "\"suspendido\": %i,\n", sus);
    seq_printf(cpufile, "\"detenido\": %i,\n", det);
    seq_printf(cpufile, "\"zombie\": %i\n", zom);
    seq_printf(cpufile, "}\n");
    seq_printf(cpufile, "}");

    return 0;
}
```

Esta función permite la lectura de los procesos que hay en el sistema por medio de las estructuras que Linux tiene por defecto

```
unsigned long rss;
si_meminfo(&inf);

int p = 0;

int tot = 0;
int eje = 0;
int sus = 0;
int det = 0;
int zom = 0;
```

Primero se inicializa un valor rss que permitirá obtener la ram que ocupa el proceso y posteriormente variables que permitirán realizar el conteo de los tipos de proceso que hay en el sistema

```
seq_printf(cpufile, "{\n\"procesos\":[\n");
for_each_process( task ){…

seq_printf(cpufile, "}\n");
seq_printf(cpufile, "],\n");
seq_printf(cpufile, "\"datos\":{\n");
seq_printf(cpufile, "\"total\": %i,\n", tot);
seq_printf(cpufile, "\"ejecucion\": %i,\n", eje);
seq_printf(cpufile, "\"suspendido\": %i,\n", sus);
seq_printf(cpufile, "\"detenido\": %i,\n", det);
seq_printf(cpufile, "\"zombie\": %i\n", zom);
seq_printf(cpufile, "}\n");
seq_printf(cpufile, "}");

return 0;
```

Después se empieza a escribir en el archivo la información que se quiere agregar y se empieza a iterar los procesos por medio de un for

```
tot++;
if(task->state == 0){
    eje++;
}else if(task->state == 1 || task->state == 2){
    sus++;
}else if(task->state == 4){
    det++;
}else if(task->exit_state == 32){
    zom++;
}
```

Dentro del for se tiene varios if que permiten determinar que tipo de proceso es el que se esta leyendo

```
seq_printf(cpufile, "{\n");
seq_printf(cpufile, "\"pid\": %d,\n",task->pid);
seq_printf(cpufile, "\"nombre\": \"%s\",\n",task->comm);
seq_printf(cpufile, "\"estado\": %ld,\n",task->state);
if (task->mm) {
    rss = get_mm_rss(task->mm) << PAGE_SHIFT;
    seq_printf(cpufile, "\"uso\": %lu,\n", rss/(1024));
    seq_printf(cpufile, "\"total\": %lu,\n", (inf.totalram * 4));
} else {
    rss = 0;
    seq_printf(cpufile, "\"uso\": %lu,\n", rss);
    seq_printf(cpufile, "\"total\": %lu,\n", (inf.totalram * 4));
}
seq_printf(cpufile, "\"mem\": %lu,\n", task->usage);
seq_printf(cpufile, "\"usuario\": \"%d\",\n", __kuid_val(task_uid(task)));
seq_printf(cpufile, "\"hijo\":\n");
seq_printf(cpufile, "\t[\n");
```

Después se escribe en el archivo la información del proceso donde se agrega el id, nombre, estado, para la memoria en uso se utilizo una función get mm rss la cual devolvía la memoria ram que ocupa el proceso, la memoria que usa el proceso, el id del usuario y un listado de hijos.

API

Una API es un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones. API significa interfaz de programación de aplicaciones.

Las API permiten que sus productos y servicios se comuniquen con otros, sin necesidad de saber cómo están implementados. Esto simplifica el desarrollo de las aplicaciones y permite ahorrar tiempo y dinero.

GO

```go
package main

import (
    "fmt"
    "log"
    "net/http"
    "encoding/json"
    "io/ioutil"
    "os/exec"
    "strconv"
)
```

Se inicializan que estará el main y se hacen las importaciones de las libreriras que se utilizaran para leer el archivo, enviar json, recibir las peticiones http

```go
func setupResponse(w *http.ResponseWriter, req *http.Request) {
    (*w).Header().Set("Access-Control-Allow-Origin", "*")
    (*w).Header().Set("Access-Control-Allow-Methods", "POST, GET, OPTIONS, PUT, DELETE")
    (*w).Header().Set("Access-Control-Allow-Headers", "Accept, Content-Type, Content-Length, Accept-Encoding, X-CSRF-Token, Authorization")
}
```

Se inicilizan las cors que permitirán que el frontend pueda obtener la información del api

```go
func homePage(w http.ResponseWriter, r *http.Request){
    setupResponse(&w, r)
    /*if (*req).Method == "OPTIONS" {
        return
    }*/

    fmt.Fprintf(w, "Welcome to the HomePage!")
    fmt.Println("Endpoint Hit: homePage")
}
```

Este endpoint se utilizara para determinar que la api funciona correctamente

```go
func ram(w http.ResponseWriter, r *http.Request){
    setupResponse(&w, r)

    bytesLeidos, err := ioutil.ReadFile("/proc/ram_p1")
    if err != nil {
        fmt.Printf("Error leyendo archivo: %v", err)
    }

    var result map[string]interface{}
    json.Unmarshal([]byte(bytesLeidos), &result)

    fmt.Println("Endpoint Hit: return ram")
    json.NewEncoder(w).Encode(result)
}
```

Ese endpoint permitirá la lectura del archivo de memoria ram que se crea con el modulo el cual se parseara a json y posteriormente se enviara como resultado del request

```go
type Hijo struct {
    Pid         int
    Nombre      string
}

type Procesos struct {
    Pid         int
    Nombre      string
    Estado      int
    Uso         int
    Total       int
    Mem         int
    Usuario     string
    Hijo        []Hijo
}

type Datos struct {
    Total       int
    Ejecucion   int
    Suspendido  int
    Detenido    int
    Zombie      int
}

type General struct {
    Procesos    []Procesos
    Datos       Datos
}
```

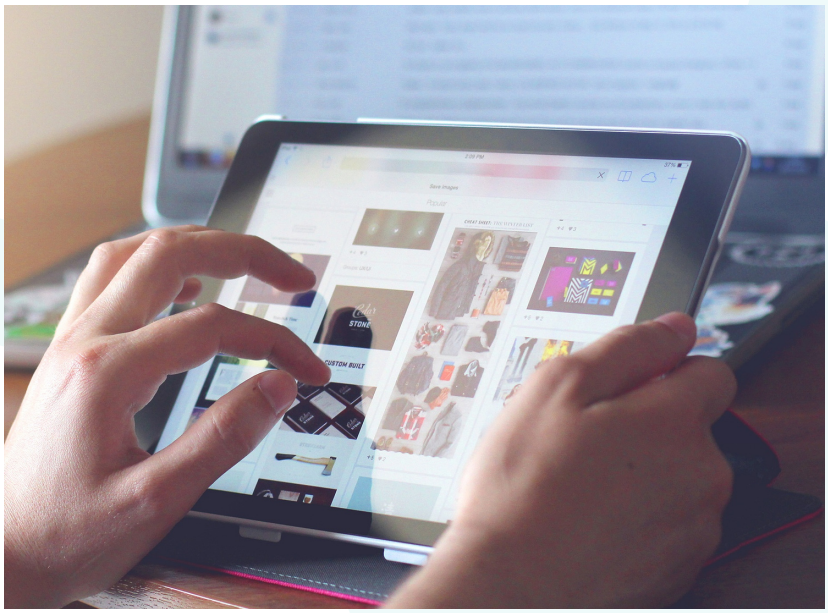
Estas estructuras se utliizaran para poder parsear la información del cpu a un json

```go
func procesos(w http.ResponseWriter, r *http.Request){
    setupResponse(&w, r)

    bytesLeidos, err := ioutil.ReadFile("/proc/cpu_p1")
    if err != nil {
        fmt.Printf("Error leyendo archivo: %v", err)
    }

    var result General
    json.Unmarshal([]byte(bytesLeidos), &result)

    /*contenido := string(bytesLeidos)
    fmt.Println(contenido)*/

    fmt.Println("Endpoint Hit: return procesos")
    //json.NewEncoder(w).Encode(contenido)
    json.NewEncoder(w).Encode(result)
    //json.NewEncoder(w).Encode("{articulo: 'llego procesos'}")
}
```

Esta función permitirá leer el archivo de los procesos que se crea con el modulo el cual se parseara a un json para posteriormente ser enviado como respuesta al request

```go
type Data struct {
    Id          int
}

type Send struct {
    Result          bool
    Id          int
}
```

Estas estructuras se utliizaran para poder parsear la información del la peticion a un json y retornar un json



```go
func kill(w http.ResponseWriter, r *http.Request) {
    setupResponse(&w, r)

    var d Data
    if json.NewDecoder(r.Body).Decode(&d) != nil {
        //println("CLIENTE: error 3")
    } else {
        //firefox id 10845 / ps -ef | grep firefox
        i := strconv.Itoa(d.Id)

        var r Send

        r.Result = muerterio(i);
        r.Id = d.Id;

        fmt.Println("Endpoint Hit: return kill")
        json.NewEncoder(w).Encode(r)
    }
}
```

Esta función permitirá obtener un id del request para posteriormente realizar la acción de matar un proceso retornando si se pudo hacer o no

```go
func muerterio(id string) bool {
    cmd := exec.Command("kill", "-9", id)
    stdout, err := cmd.StdoutPipe()
    if  err != nil {// Obtenga el objeto de salida, puede leer el resultado de salida del objeto
        fmt.Println(err)
    }

    defer stdout.Close () // Garantía para cerrar el flujo de salida

    if err := cmd.Start(); err != nil {// Ejecutar comando
        fmt.Println(err)
    }

    opBytes, err := ioutil.ReadAll(stdout);
    if err != nil {// Leer el resultado de salida
        fmt.Println(err)
        return false
    } else {
        fmt.Println(string(opBytes))
        return true
    }
}
```

Esta función realiza por medio del cmd la ejecución del comando kill que permite matar un proceso

```go
func handleRequests() {
    fmt.Println("Servidor escuchando en 8080")
    http.HandleFunc("/", homePage)
    http.HandleFunc("/ram", ram)
    http.HandleFunc("/cpu", procesos)
    http.HandleFunc("/kill", kill)
    log.Fatal(http.ListenAndServe(":8080", nil))
}

func main() {
    handleRequests()
}
```

Estas ultimas ufnciones permiten crear el ervidor y se agregan los endpoins llamando las funciones que utlizara cada uno

#Listar procesos

```
ps

#Version de kernel
uname -a

#Instalar
sudo apt-get install make gcc

#Compilar el c
make

#Ver los modlos
lsmod

#Ver los archivos en lista
ls -l

#Agregar el modulo
insmod modulo_cpu.ko

#Eliminar modulo
rmmod modulo_cpu

#Ver modulo existente
lsmod | grep modulo_cpu
```

#RAM

```
dmesg -C
make clean
rmmod ram_p1
make
insmod ram_p1.ko
cat /proc/ram_p1
dmesg
ls -l ram_p1
```

#CPU

```
dmesg -C
make clean
rmmod cpu_p1
make
insmod cpu_p1.ko
cat /proc/cpu_p1
dmesg
ls -l /proc/cpu_p1
```