MANUAL DE CREACIÓN DE MODULOS: Ubuntu 18.04

Módulo de RAM y CPU: estos fragmentos de código son utilizados en ambos módulos.

Para el módulo de RAM es necesario crear 2 llamadas, uno para iniciar el módulo y el otro para salir del módulo, estas llamadas son:

```
// esta llamada carga la función que se ejecutará en el init
module_init(on_init);

// esta llamada carga la función que se ejecutará en el exit
module_exit(on_exit);
```

On_init: este método indica el código que se ejecutará cuando montemos el modulo. Aquí definimos el archivo que ira dentro de la carpeta proc, este se llama memo_201603157 si esta operación falla el resto de código no se ejecuta de lo contrario escribirá que el modulo esta siendo montado.

```
// Definicion de evento principal
static int __init on_init(void)
{
    struct proc_dir_entry *entry;

        entry = proc_create("memo_201603157",0777,NULL,&operaciones);

    if(!entry){
                return -1;
        }else{
                printk(KERN_INFO "Creando proc file... carnet 201603157\n");
        }

        return 0;
}
```

On_exit: este método remueve el archivo memo_201603157 de la carpeta /proc.

```
static void __exit on_exit(void)
{
    // Código dentro del evento EXIT
    remove_proc_entry("memo_201603157", NULL);
    printk(KERN_INFO "Removiendo modulo ram Sistemas Opearativos 1\n");
}
```

En el método on_init utilizamos un puntero llamado operaciones el cual viene del struct file_operations este struct se encargara de la escritura en nuestro archivo memo_201603157.

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

De este struct las partes mas importantes son .open y .write, my_proc_open y my_proc_write respectivamente, estos metodos se encargaran de abrir y escribir en el archivo memo_201603156 cuando se ejecute el comando cat.

```
static ssize_t my_proc_write(struct file* file,const char __user *buffer,size_t count,loff_t *f_pos){
        char *tmp = kzalloc((count+1),GFP_KERNEL);
        if(!tmp)return -ENOMEM;
        if(copy_from_user(tmp,buffer,count)){
                kfree(tmp);
                return EFAULT;
        }
        kfree(str);
        str=tmp;
        return count;
}

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

Módulo RAM:

El modulo ram tiene un método llamado my_proc_show donde se ejecuta los mas importante que es donde utilizamos el struct sysinfo el cual no brinda toda la información de la memoria ram, para calcular los valores exactos en MB se hizo la siguiente operación x * 4 / 1024 donde x es el valor que necesitamos en el modulo, por ejemplo: si.totalram nos da el total de RAM de nuestro ordenador, si.freeram nos brinda la ram que no esta siendo utilizada y si.sharedram es la memoria compartida. Lo que se busca es tener una salida en formato json el cual será fácilmente leído y parseado a un struct en el servidor de golang.

```c
static int my_proc_show(struct seq_file *archivo,void *v){

    //const unsigned long megabyte = 1024 * 1024;

    struct sysinfo si;
    si_meminfo (&si);
    //unsigned long totalDeRam = si.totalram;
    //unsigned long ramLibre = si.freeram;
    int number = 10;

    //seq_printf(archivo, "[\n");
    seq_printf(archivo, "{");
    seq_printf(archivo, "\"totalRam\":%lu,", si.totalram * 4 / 1024);
    seq_printf(archivo, "\"freeRam\":%lu,", si.freeram * 4 / 1024);
        seq_printf(archivo, "\"sharedRam\":%lu,", si.sharedram * 4 / 1024);
    seq_printf(archivo, "\"usageRam\":%lu,", (si.totalram - si.freeram) * 4 / 1024);
    seq_printf(archivo, "\"usagePercentage\":%lu", ((si.totalram - si.freeram) * 100) / si.totalram);
    seq_printf(archivo, "}");
    //seq_printf(archivo, "]\n");

        return 0;
}
```

Módulo CPU:

Para el módulo de CPU utilizamos 2 structs importantes, task_struct el cual nos brinda la información de todos los procesos que están siendo ejecutados y para ello se crean dos punteros de este struct uno para el proceso padre (task) y el otro para el proceso hijo (task_child), el otro struct que utilizamos es el list_head que nos ayudará a recorrer todos los hijos que tenga un proceso padre (como una lista de hijos) y la idea de este método es formar un array en formato json donde cada objeto dentro contiene un proceso del CPU y a su vez contiene un array de objetos hijos, esto es parseado por un método dentro del servidor de golang.

```
for_each_process( task ){

    if(contador != 0) {
        seq_printf(archivo, ",");
    }

    contadorHijos = 0;

    seq_printf(archivo, "{");
    seq_printf(archivo, "\"pid\": %d,", task->pid);
    seq_printf(archivo, "\"name\": \"%s\",", task->comm);
    seq_printf(archivo, "\"state\": %ld,", task->state);
    seq_printf(archivo,  "\"uid\": %d,", __kuid_val(task->real_cred->uid));
    seq_printf(archivo, "\"childs\": [");
    //seq_printf(archivo, "PARENT PID: %d PROCESS: %s STATE: %ld",task->pid,
    list_for_each(list, &task->children){                        /*    list_f

        if(contadorHijos != 0) {
            seq_printf(archivo, ",");
        }

        task_child = list_entry( list, struct task_struct, sibling );    /*

        seq_printf(archivo, "{");
        seq_printf(archivo, "\"pid\": %d,", task_child->pid);
        seq_printf(archivo, "\"name\": \"%s\",", task_child->comm);
        seq_printf(archivo, "\"state\": %ld", task_child->state);

        seq_printf(archivo, "}");

        contadorHijos++;
    }
    seq_printf(archivo, "]");
    seq_printf(archivo, "}");
    contador++;
}
```

Creando módulos:

Para montar los modulos utilizamos el comando make all en la carpeta donde se encuentran nuestros archivos .c y Makefile

Mondando módulos:

Para montar el modulo ejecutamos el comando: sudo insmod <nombre_modulo>.ko

Para leer el comando: cat /proc/<nombre_modulo>