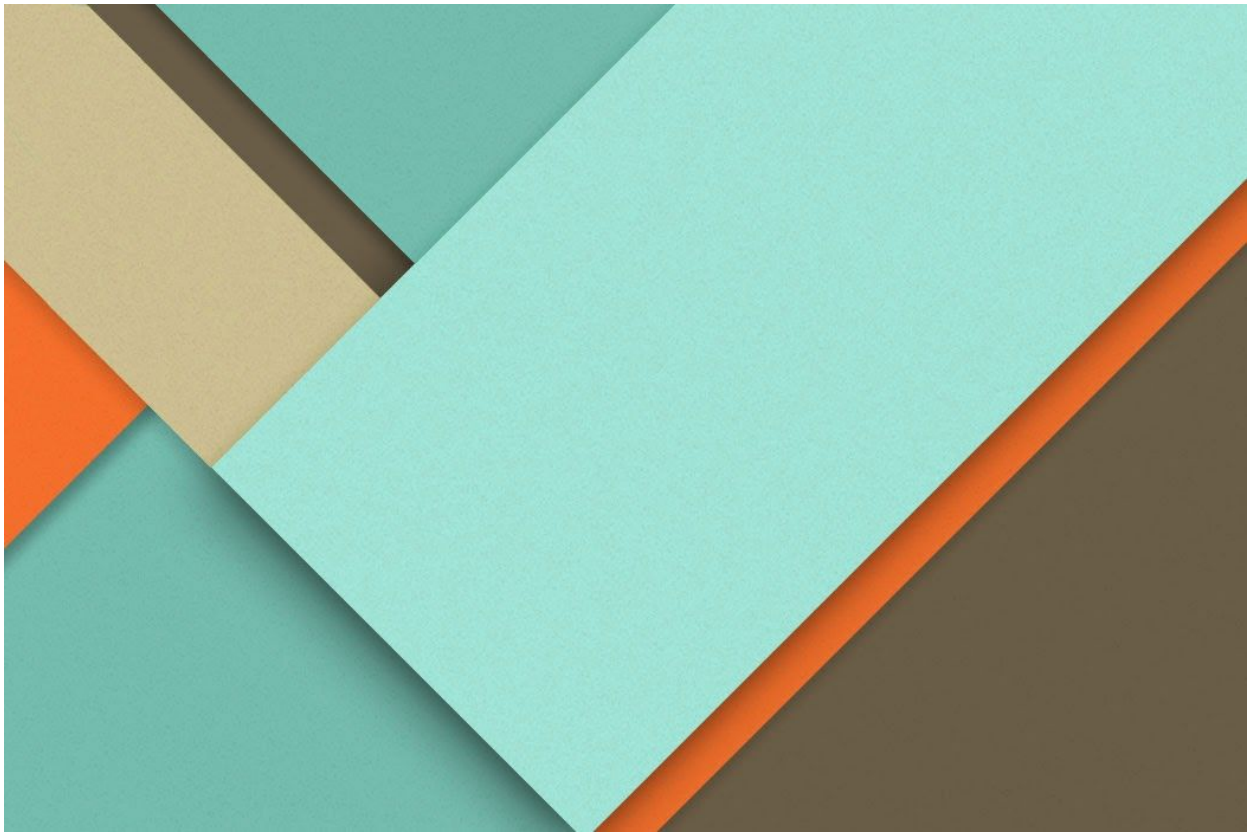

Creacion de modulos

Jorge Mario Recinos Dieguez
201314631
Sistemas Operativos 1
Diciembre 2020

Para crear los módulos debemos utilizar un archivo hecho en C. para el primer módulo de procesos utilizaremos el nombre de cpu_201314631

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/list.h>
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/sched.h>
#include <linux/seq_file.h>
#include <linux/slab.h>
#include <linux/string.h>
#include <linux/types.h>
#include <linux/mm.h>
```

Para ambos módulos debemos hacer el uso de las librerías. Estos se encuentran en los headers de Linux para nuestro kernel. Por eso es importante tenerlos instalados. Seguimos con una pequeña descripción del módulo del kernel a modo de metadata.

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Jorge Mario Recinos Dieguez");
MODULE_DESCRIPTION("Modulo de CPU - Sistemas Operativos 1");
```

Y debemos crear dos funciones. Uno para abrir un archivo y otra función para poder escribir los datos que vamos a monitorear desde el kernel.

```
static int escribir_archivo(struct seq_file * archivo, void *v)
static int abrirArchivo(struct inode *inode, struct  file *file)
```

La información la obtendremos de un struct ya declarado en nuestras importaciones.

```
struct sysinfo inf;
```

Ya con nuestra estructura podemos empezar a escribir el cuerpo de nuestro método de escribir el archivo.

```
si_meminfo(&inf);
long total_memoria = (inf.totalram * 4);
long memoria_libre = (inf.freeram * 4 );
long memBuffer = (inf.bufferram*4);
long memCache = (inf.sharedram*4);
```

Podemos obtener el total de RAM, la memoria libre y lo que se encuentra en cache y buffer. Todo desde esta estructura. Empezamos a crear nuestros archivos.

```
seq_printf(archivo, "{");
   seq_printf(archivo, "\"mem_total\":%8lu,\n",total_memoria);
   seq_printf(archivo, "\"mem_free\":%8lu,\n", memoria_libre);
   seq_printf(archivo, "\"mem_inuse\":\"%li%%\",\n", ((total_memoria - memoria_libre) * 100)/total_memoria) ;
   seq_printf(archivo, "\"mem_buffer\":%li,\n", memBuffer);
   seq_printf(archivo, "\"mem_cache\":%li\n", memCache);
   seq_printf(archivo, "}");
```

Ahora que ya tenemos nuestro archivo escrito. Lo que hacemos es mandarlo a crear desde nuestra función de abrir archivo.

```
return single_open(file, escribir_archivo, NULL);
```

Ya definidas nuestras funciones la declaramos en el lenguaje para que sean ejecutadas.

```
static struct file_operations operaciones =
{
.open = abrirArchivo,
```

```
.read = seq_read
};
```

Luego debemos crear el proceso al ser montado. Y creamos un pequeño mensaje de bienvenida.

```
static int al_iniciar(void)
{
    proc_create("mem_201314631", 0, NULL, &operaciones);
    printk(KERN_INFO "          201314631     \n");
    return 0;
}
```

Como también se creó otro método con la que le indicaremos que debe desmontar el proceso y además agregamos un mensaje de salida.

```
static void al_salir(void)
{
    remove_proc_entry("mem_201314631", NULL);
    printk(KERN_INFO "        Sistemas Operativos 1    \n");
}
```

Por último, debemos especificarle cuales son los métodos y funciones que utilizamos para la salida y creación del módulo respectivamente.

```
module_init(al_iniciar);
module_exit(al_salir);
```

Compilacion de Módulos

Para compilar este módulo ya finalizado de monitor de procesos. Debemos crear un archivo sin extensión llamando Makefile. Este es utilizado por la herramienta MAKE para compilar nuestro modulo.

```
obj-m += mem_201314631.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

A este archivo le indicamos dos comandos uno que es all y el otro clean. En el all es para que genere todo los módulos a partir del archivo en C en la misma ruta llamado mem_201314631.c.

Así como el otro comando que elimina o borra todos los binarios generados. Este nos referimos al comando clean.

Para el módulo de proceso se sigue la misma dinámica. Solo que ahora los valores los obtenemos de otra estructura.

```
void readProcess(struct seq_file *m, struct task_struct *s){
struct list_head *list;
struct task_struct *task;
```

En estas dos estructuras obtendremos toda la lista de procesos con la cual crearemos el árbol de procesos.

Luego tenemos un switch que nos ayuda a identificar el código de estado de cada proceso. Este con sus valores ya definidos con las palabras reservadas TASK_RUNNING, TASK_STOPPED, TASK_INTERRUPTIBLE, etc.

```
switch(s->state){
case TASK_RUNNING:
strcpy(estado,"Ejecucion");
break;
case TASK_STOPPED:
strcpy(estado,"Detenido");
break;
case TASK_INTERRUPTIBLE:
strcpy(estado,"Interrumpible");
break;
case TASK_UNINTERRUPTIBLE:
strcpy(estado,"Ininterrumpible");
break;
case EXIT_ZOMBIE:
strcpy(estado,"Zombi");
break;
default:
strcpy(estado, "Desconocido");
}
```

Por último hacemos el Split de todos los procesos hijos.

```
seq_printf(m,"{\n");
seq_printf(m,"PID: %d,\t\t\nNombre: %s,\t\t\nEstado:%s,\t\t \nUsuario:%d,\t\t
\nMemoria:%d\n\n\n",s->pid, s->comm, estado, s->mm, __kuid_val(s->real_cred-
```

```
>uid) );
seq_printf(m,"Children:[\n");
list_for_each(list, &s->children) {
task = list_entry(list, struct task_struct, sibling);
readProcess2(m, task);
}
```

Donde s->pid es el identificador del proceso

s->comm es el nombre del proceso

s->mm es el valor de memoria utilizado por el proceso en.

__kuid_val(s->real_cred->uid) es el identificador del usuario