Módulo de memoria

1) Importar las siguientes librerías en el archivo C

```
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <asm/uaccess.h>
#include <linux/hugetlb.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
```

2) Definimos el tamaño del buffer y colocamos una descripción del módulo

```
#define BUFSIZE 150

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Monitor de ram");
MODULE_AUTHOR("David González");
```

3) Colocamos el código principal de nuestro módulo. Donde:
seq_file: Es un struct del sistema que permite escribir un archivo
sysinfo: Es el estruct que contiene la información de la memoria del sistema.

```
struct sysinfo info;

static int infoMemoria(struct seq_file *arch, void *v){

	unsigned long megas=1024*1024;
	unsigned long total = 0;
	unsigned long libre = 0;
	unsigned long unidad = 0;

	seq_printf(arch, "************************************\
n");
	seq_printf(arch, "*     Carné:   201610648           *\
n");
	seq_printf(arch, "*     Nombre:  David González      *\
n");
	seq_printf(arch, "*----------------------------------*\
n");
	seq_printf(arch, "*     Carné:   201403841           *\
n");
```

```
        seq_printf(arch, "*     Nombre:  Huriel Gómez        *\
n");
        seq_printf(arch, "************************************\
n");
        seq_printf(arch, "*         MODULO DE MEMORIA        *\
n");
        seq_printf(arch, "************************************\
n");
```

4) **si_meminfo(* struct sysinfo):** Es un método del sistema que permite llenar el structo sysinfo con la información del sistema.

   info.mem_unit: Contiene la unidad en la que esta dada la informacion (bytes)
   info.totalram: Contiene el total de ram en nuestro sistema
   info.freeram: Contiene la memoria libre en nuestro sistema
   info.bufferram: Contiene la memoria en buffer.

   La información de memoria se debe de multiplicar por la info.mem_unit para que los datos sean reales.

```
        si_meminfo(&info);
        unidad = (unsigned long)info.mem_unit;
        total = info.totalram * unidad;
        total = total/megas;
        libre = (info.freeram + info.bufferram) * unidad;
        libre = libre/megas;
        seq_printf(arch, "Carné:\t201610648\n");
        seq_printf(arch, "Nombre:\tDavid González\n");
        seq_printf(arch, "Memoria Total:\t%lu MB\n", total);
        seq_printf(arch, "Memoria Libre:\t%lu MB\n", libre);
        seq_printf(arch, "Memoria usada:\t%lu %%\n", ((total-
libre)*100)/total);

        return 0;
}
```

5) Como queremos que la información se actualice al abrir el archivo, configuramos los eventos:

   Se usan los siguientes métodos que tienen que tener la siguiente estructura:
   evento_abrir: En el retorno se debe de pasar como parámetro nuestro método creado que contiene el código a ejecutar (infoMemoria)

   file_operations: Este cuenta con un atributo **open**: que apunta al método evento_abrir, un atributo **read** que apunta al tipo de operación que deseamos realizar

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

static struct file_operations operaciones = {
	.open=evento_abrir,
	.read = seq_read
};
```

6) **inicio**: Este método se ejecuta al insertar el módulo.
   fin: Este método se ejecuta al dar de baja el módulo
   module_init: Apunta al método que se ejecuta al insertar el módulo.
   module_exit: Apunta al metodo que se ejecuta al dar de baja el módulo

```
static int inicio(void)
{
	proc_create("memo_201610648_201403841", 0, NULL,
&operaciones);
	printk(KERN_INFO "Carne: 201610648 -- 201403841\n");
	return 0;
}

static void fin(void)
{
	remove_proc_entry("memo_201610648_201403841", NULL);
	printk(KERN_INFO "Curso: Sistemas Operativos 1\n");
}

module_init(inicio);
module_exit(fin);
```

Módulo de cpu

1) Importar las siguientes librerías en el archivo C

```c
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <asm/uaccess.h>
#include <linux/hugetlb.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/sched/signal.h>
#include <linux/sched.h>
```

2) Definimos el tamaño del buffer y colocamos una descripción del módulo

```c
#define BUFSIZE 150

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Monitor de cpu");
MODULE_AUTHOR("David González");
```

3) Colocamos el código principal de nuestro módulo. Donde:
seq_file: Se utiliza para escribir el archivo
task_struct: Contiene la información del proceso y sus hijos
list_head: Contiene la lista de procesos

```c
static int infoMemoria(struct seq_file *arch, void *v)
{
	struct task_struct *ts;
	struct task_struct *hijos;
	struct list_head *list;
	seq_printf(arch, "*********************************\n");
	seq_printf(arch, "*     Carné:   201610648          *\n");
	seq_printf(arch, "*     Nombre:  David González     *\n");
	seq_printf(arch, "*---------------------------------*\n");
	seq_printf(arch, "*     Carné:   201403841          *\n");
	seq_printf(arch, "*     Nombre:  Huriel Gómez       *\n");
	seq_printf(arch, "*********************************\n");
	seq_printf(arch, "*           MODULO CPU            *\n");
	seq_printf(arch, "*********************************\n");
```

for_each_procces(struct *task_struct): Se usa para recorrer los procesos y almacenar a cada vuelta de bucle en el task_struct

list_for_each(struct list_head, struct task_struct -> children): Toma los hijos del task_struct actual y lo coloca en el list head, luego con el metodo **list_entry(struct list_head, struct task_struct, sibiling),** se obtiene cada proceso hijo y retorna su task_struct para obtener la información.

```
	for_each_process(ts)
	{
		//seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts->comm,
ts->state);

		if (ts->state == 0)
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts-
>comm, "Task Running");
		}
		else if (ts->state == 1)
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts-
>comm, "Task Interruptible");
		}
		else if (ts->state == 2)
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts-
>comm, "Task Uninterruptible");
		}
		else if (ts->state == 4)
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts-
>comm, "Task Zombie");
		}
		else if (ts->state == 8)
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %s \n", ts->pid, ts-
>comm, "Task Stopped");
		}
		else
		{
			seq_printf(arch, "\nPID: %d | NOMBRE: %s | ESTADO: %ld \n", ts->pid,
ts->comm, ts->state);
		}

		list_for_each(list, &ts->children)
		{
			hijos = list_entry(list, struct task_struct, sibling);
			if (hijos->state == 0)
			{
				seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %s \n",
hijos->pid, hijos->comm, "Task Running");
			}
			else if (hijos->state == 1)
			{
				seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %s \n",
```

```
hijos->pid, hijos->comm, "Task Interruptible");
                        }
                        else if (hijos->state == 2)
                        {
                                seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %s \n",
hijos->pid, hijos->comm, "Task Uninterruptible");
                        }
                        else if (hijos->state == 4)
                        {
                                seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %s \n",
hijos->pid, hijos->comm, "Task Zombie");
                        }
                        else if (hijos->state == 8)
                        {
                                seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %s \n",
hijos->pid, hijos->comm, "Task Stopped");
                        }
                        else
                        {
                                seq_printf(arch, "|____PID: %d | NOMBRE: %s | ESTADO: %ld \n",
hijos->pid, hijos->comm, hijos->state);
                        }
                }
        }

        return 0;
}
```

4) Como queremos que la información se actualice al abrir el archivo, configuramos los eventos:

   Se usan los siguientes métodos que tienen que tener la siguiente estructura: **evento_abrir**: En el retorno se debe de pasar como parámetro nuestro método creado que contiene el código a ejecutar (infoMemoria)

   file_operations: Este cuenta con un atributo **open**: que apunta al método evento_abrir, un atributo **read** que apunta al tipo de operación que deseamos realizar

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

static struct file_operations operaciones = {
        .open=evento_abrir,
        .read = seq_read
};
```

5) **inicio**: Este método se ejecuta al insertar el módulo.
fin: Este método se ejecuta al dar de baja el módulo
module_init: Apunta al método que se ejecuta al insertar el módulo.
module_exit: Apunta al metodo que se ejecuta al dar de baja el módulo

```
static int inicio(void)
{
	proc_create("memo_201610648_201403841", 0, NULL,
&operaciones);
	printk(KERN_INFO "Carne: 201610648 -- 201403841\n");
	return 0;
}

static void fin(void)
{
	remove_proc_entry("memo_201610648_201403841", NULL);
	printk(KERN_INFO "Curso: Sistemas Operativos 1\n");
}

module_init(inicio);
module_exit(fin);
```