

Universidad De San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Vacaciones Diciembre 2021

SISTEMAS OPERATIVOS 1
SECCIÓN "A"



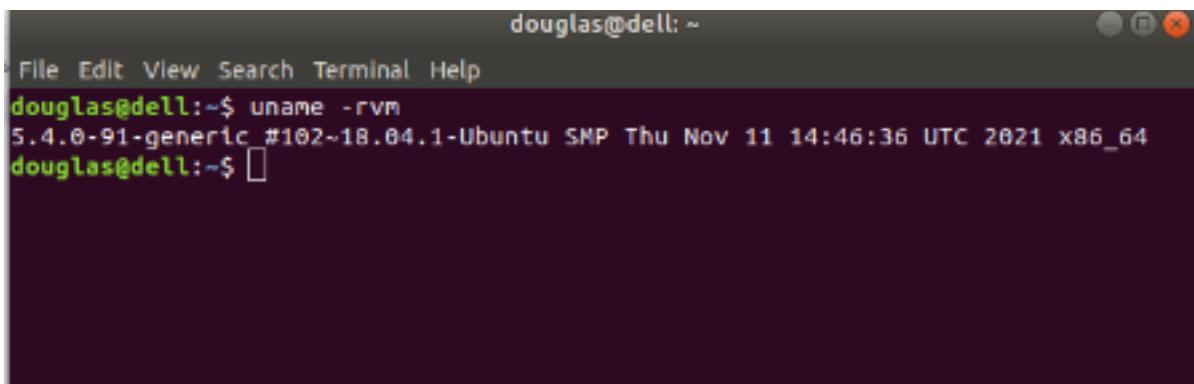
CONFIGURACION DE MODULOS

Douglas Omar Arreola Martínez

Carné: 201603168

DEPENDENCIAS

Antes de crear los módulos kernel solicitados debemos preparar el entorno del sistema operativo para que permita crear y configurar nuevos módulos a partir de código escrito en lenguaje C. Para ello se debe de instalar los headers para desarrollo del núcleo específico de kernel que tiene la computadora los cuales no están incluidos de forma predeterminada en las distribuciones linux.



```
douglas@dell: ~
File Edit View Search Terminal Help
douglas@dell:~$ uname -rvm
5.4.0-91-generic #102~18.04.1-Ubuntu SMP Thu Nov 11 14:46:36 UTC 2021 x86_64
douglas@dell:~$ █
```

A screenshot of a terminal window titled "douglas@dell: ~". The window has a dark background with light-colored text. It shows the user's name "douglas", the host name "dell", and the current directory "~". Below the title bar is a menu bar with options: File, Edit, View, Search, Terminal, and Help. The main area of the terminal displays the command "uname -rvm" followed by its output. The output shows the kernel version "5.4.0-91-generic", the kernel revision "#102~18.04.1-Ubuntu", the build type "SMP", the build date "Thu Nov 11 14:46:36 UTC 2021", and the architecture "x86_64". The terminal window has a standard window control bar at the top right with minimize, maximize, and close buttons.

Con el comando **uname -r** se verifica la versión de Kernel presente en la computadora.

En el caso de Debian y Ubuntu (el cual es mi caso) estas dependencias se obtienen con los siguientes comandos:

- Para descargar headers del módulo específico que tenemos:

```
$ sudo apt-get install linux-headers-$(uname -r)
```

- Descargar build essentials, para compilar el código escrito en lenguaje C:

```
$ sudo apt-get install build-essential
```

MÓDULO RAM

Librerías en Uso

Para el desarrollo del módulo de monitoreo de memoria RAM se utilizaron las siguientes librerías mostradas en la imagen de la derecha; de las cuales ***module.h***, ***kernel.h*** e ***init.h*** proporcionan las funciones necesarias para poder compilar el código de C así como los medios necesarios para agregar el nuevo módulo al directorio **/proc/**. También las librerías ***fs.h***, ***seq_file.h*** y ***proc_fs.h*** fueron utilizadas para que se pueda crear el archivo de tipo **.ko (Kernel Object)** el cual es utilizado para luego crear el módulo.

```
1 #include <linux/sysinfo.h>
2 #include <linux/module.h>
3 #include <linux/kernel.h>
4 #include <linux/fs.h>
5 #include <linux/mm.h>
6 #include <linux/seq_file.h>
7 #include <linux/proc_fs.h>
8 #include <linux/init.h>
9 |
```

Para la codificación principal del módulo se utilizó la librería ***sysinfo.h*** para obtener la información de hardware y posteriormente escribir los datos solicitados en el archivo del módulo kernel.

Información de la Estructura

La estructura utilizada para obtener y almacenar los datos del sistema (incluida la memoria RAM) se llama ***sysinfo*** y provee la siguiente información, dada por la documentación de la libreria ***linux/sysinfo.h*** disponible online en el sitio web: <https://man7.org/linux/man-pages/man2/sysinfo.2.html>.

Mi versión del kernel de linux es superior a la 2.3.23 así que utilice el struct mostrado en la parte inferior de la imagen siguiente, de ellos los principales atributos son ***totalram***, ***freeram***, ***sharedram*** y ***mem_unit*** que como dicen sus nombres en inglés nos dan la información principal respecto al estado de la memoria RAM del sistema en un tiempo específico (todo dato está dado en bytes).

```

DESCRIPTION      top

    sysinfo() returns certain statistics on memory and swap usage, as
    well as the load average.

    Until Linux 2.3.16, sysinfo() returned information in the
    following structure:

    struct sysinfo {
        long uptime;          /* Seconds since boot */
        unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
        unsigned long totalram; /* Total usable main memory size */
        unsigned long freeram; /* Available memory size */
        unsigned long sharedram; /* Amount of shared memory */
        unsigned long bufferram; /* Memory used by buffers */
        unsigned long totalswap; /* Total swap space size */
        unsigned long freeswap; /* Swap space still available */
        unsigned short procs; /* Number of current processes */
        char _f[22];           /* Pads structure to 64 bytes */
    };

    In the above structure, the sizes of the memory and swap fields
    are given in bytes.

    Since Linux 2.3.23 (i386) and Linux 2.3.48 (all architectures)
    the structure is:

    struct sysinfo {
        long uptime;          /* Seconds since boot */
        unsigned long loads[3]; /* 1, 5, and 15 minute load averages */
        unsigned long totalram; /* Total usable main memory size */
        unsigned long freeram; /* Available memory size */
        unsigned long sharedram; /* Amount of shared memory */
        unsigned long bufferram; /* Memory used by buffers */
        unsigned long totalswap; /* Total swap space size */
        unsigned long freeswap; /* Swap space still available */
        unsigned short procs; /* Number of current processes */
        unsigned long totalhigh; /* Total high memory size */
        unsigned long freehigh; /* Available high memory size */
        unsigned int mem_unit; /* Memory unit size in bytes */
        char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding to 64 bytes */
    };

    In the above structure, sizes of the memory and swap fields are
    given as multiples of mem_unit bytes.

```

Atributos de la estructura **sysinfo** para núcleos del kernel Linux previos y posteriores a la versión 2.3.16

Funciones Usadas

El funcionamiento principal del módulo kernel está dado por estas funciones que indican al sistema que debe de hacer cuando el módulo es montado (**module_init**) y desmontado (**module_exit**) por el usuario.

```

> static int __init ram_read_init(void) ...
> static void __exit ram_read_exit(void) ...

    module_init(ram_read_init);
    module_exit(ram_read_exit);

```

Posteriormente para indicar que debe de hacer cada vez que se lea el proceso con el comando “`cat /proc/memo_201603168`” se utilizan las funciones `write_file_proc`, `open_file_proc`, `single_open` y todas ellas se asignan en el struct `file_operations` que es el que desde el momento de compilación del código indica que debe de hacer cuando el módulo es leído, abierto, escrito (en caso de que sea necesario) etc.

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

static int open_file_proc(struct inode *inode, struct file *file)
{
    return single_open(file, show_ram_data, NULL);
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = open_file_proc,
    .release = single_release,
    .read = seq_read,
    .llseek = seq_llseek,
    .write = write_file_proc
};
```

Por último en la función `show_ram_data` se hace una llamada al método `si_meminfo` que es el que lee la información del hardware y la guarda en el puntero del struct `sysinfo` para luego que el proceso o módulo de imprima de forma secuencial la información pertinente bajo un formato JSON.

```
static int show_ram_data(struct seq_file *m, void *v)
{
    #define K(x) ((x) << (PAGE_SHIFT - 10))

    si_meminfo(&si);
    seq_printf(m, "{\"TOTAL\":%8lu,\"FREE\":%8lu,\"SHARED\":%8lu}", K(si.totalram), K(si.freeram), K(si.sharedram));
    return 0;
}
```

Comandos en consola:

Con el objetivo de compilar el código escrito en C y que el objeto `.ko` sea montado en el kernel se utiliza un archivo **Make** el cual contiene las instrucciones y/o comandos necesarios para automatizar este proceso. Además, de manera personal, se agregaron más comandos los cuales ayudan a montar y desmontar el módulo en cuestión de una forma más rápida y simple.

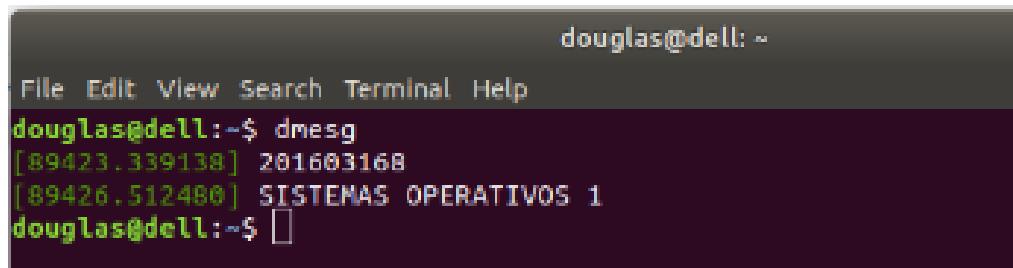
```

ram_module > Makefile
1  obj-m := memo_201603168.o
2
3  KDIR := /lib/modules/$(shell uname -r)/build
4
5  all:
6      $(MAKE) -C $(KDIR) M=$(shell pwd)
7
8  clean:
9      $(MAKE) -C $(KDIR) M=$(shell pwd) clean
10
11 up:
12     @echo "CARNET: 201603168"
13     @sudo insmod memo_201603168.ko
14
15 down:
16     @echo "CURSO: SISTEMAS OPERATIVOS 1"
17     @sudo rmmod memo_201603168.ko

```

- **all:** compila el código en C y prepara los código objeto en la librería indicada en la variable **KDIR**.
- **make clean:** limpia los archivos generados con el comando make indicados en el directorio **KDIR**.
- **make up:** ejecuta el comando insmod con el que se monta el módulo en el kernel de la distribución de linux. También escribe en consola la frase “CARNET: 201603168”.
- **make down:** ejecuta el comando rmmod con el que se desmonta el módulo del kernel. También escribe en la consola la frase “CURSO: SISTEMAS OPERATIVOS 1”.

Con el comando **dmesg** se puede ver los mensajes escritos por los módulos en el buffer de mensajes del núcleo. Al montar y desmontar el modulo **memo_201603168** se pueden ver los siguientes mensajes:



The screenshot shows a terminal window with a dark background and light-colored text. At the top, it says "douglas@dell: ~". Below that is a menu bar with "File Edit View Search Terminal Help". The main area of the terminal shows the command "dmesg" being run, followed by two messages from the kernel log:

```

douglas@dell:~$ dmesg
[89423.339138] 201603168
[89426.512480] SISTEMAS OPERATIVOS 1
douglas@dell:~$ 

```

MÓDULO CPU

Librerías en Uso

Para el desarrollo del módulo de monitoreo de monitoreo de Procesos se utilizaron las siguientes librerías vistas en la imagen de la derecha; de las cuales ***module.h***, ***kernel.h*** e ***init.h*** proporcionan las funciones necesarias para poder compilar el código de C así como los medios necesarios para agregar el nuevo módulo al directorio **/proc/**. También las librerías ***fs.h***, ***seq_file.h*** y ***proc_fs.h*** fueron utilizadas para que se pueda crear el archivo de tipo **.ko (Kernel Object)** el cual es utilizado para luego crear el módulo.

```
1 #include <linux/kernel.h>
2 #include <linux/module.h>
3 #include <linux/init.h>
4 #include <linux/fs.h>
5 #include <linux/seq_file.h>
6 #include <linux/proc_fs.h>
7 #include <linux/sched.h>
8 #include <linux/mm.h>
9 #include <linux/sched/signal.h>
10 #include <linux/sysinfo.h>
11 |
```

Para la codificación principal del módulo se utilizó la librería ***sched.h*** para obtener la información de hardware, ***sched/signal.h*** para apoyar con funciones y métodos para recorrer procesos a la librería mencionada anteriormente y también las librerías ***mm.h*** y ***sysinfo.h*** para obtener los datos de memoria de cada proceso y operarlos para obtener el porcentaje de utilización de cada uno de ellos.

Información de la Estructura

La estructura utilizada para obtener y almacenar los datos de la planificación y mantenimiento del CPU (incluida los procesos y subprocessos) se llama ***task_struct*** y provee la información de los procesos que en un momento dado aparecen en la lista de procesos del sistema, la documentación de la estructura está disponible online en el sitio web: https://www.cs.fsu.edu/~baker/opsys/examples/task_struct.html. De la misma forma se utiliza otro struct llamado ***list_head*** que sirve para manejar los procesos hijos de los procesos obtenidos en la estructura ***task_struct***.

De todos los atributos disponibles se utilizan: ***pid***, ***comm***, ***state*** y ***real_cred*** para obtener respectivamente el identificador del proceso, su nombre, su estado y el usuario que lo ejecutó.

```

struct task_struct {
    /*
     * offsets of these are hardcoded elsewhere - touch with care
     */
    volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags;      /* per process flags, defined below */
    int sigpending;
    mm_segment_t addr_limit; /* thread address space:
                                0-0xFFFFFFFF for user-thread
                                0-0xFFFFFFFF for kernel-thread
                                */
    struct exec_domain *exec_domain;
    volatile long need_resched;
    unsigned long ptrace;

    int lock_depth;           /* Lock depth */

    /*
     * offset 32 begins here on 32-bit platforms. We keep
     * all fields in a single cacheline that are needed for
     * the goodness() loop in schedule().
     */
    long counter;
    long nice;
    unsigned long policy;
    struct mm_struct *mm;
    int processor;
    /*
     * cpus_runnable is ~0 if the process is not running on any
     * CPU. It's (1 << cpu) if it's running on a CPU. This mask
     * is updated under the runqueue lock.
     *
     * To determine whether a process might run on a CPU, this
     * mask is AND-ed with cpus_allowed.
     */
    unsigned long cpus_runnable, cpus_allowed;
    /*
     * (only the 'next' pointer fits into the cacheline, but
     * that's just fine.)
     */
    struct list_head run_list;
    unsigned long sleep_time;

    struct task_struct *next_task, *prev_task;
    struct mm_struct *active_mm;
    struct list_head local_pages;
    unsigned int allocation_order, nr_local_pages;

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_code, exit_signal;
    int pdeath_signal; /* The signal sent when the parent dies */
    /* ??? */
    unsigned long personality;
    int did_exec:1;
    pid_t pid;
    pid_t pgrp;
    pid_t tty_old_pgrp;
    pid_t session;
    pid_t tgid;
    /* boolean value for session group leader */
    int leader;
    /*
     * pointers to (original) parent process, youngest child, younger sibling,
     * older sibling, respectively. (p->father can be replaced with

```

Vista de la información disponible de la estructura **task_struct**.

Funciones Usadas

El funcionamiento principal del módulo kernel está dado por estas funciones que indican al sistema que debe de hacer cuando el módulo es montado (**module_init**) y desmontado (**module_exit**) por el usuario.

```
> static int __init cpu_read_init(void) ...
> static void __exit cpu_read_exit(void) ...
...
module_init(cpu_read_init);
module_exit(cpu_read_exit);
```

Posteriormente para indicar que debe de hacer cada vez que se lea el proceso con el comando “`cat /proc/cpu_201603168`” se utilizan las funciones `write_file_proc`, `open_file_proc`, `single_open` y todas ellas se asignan en el struct `file_operations` que es el que desde el momento de compilación del código indica que debe de hacer cuando el módulo es leído, abierto, escrito (en caso de que sea necesario) etc.

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

static int open_file_proc(struct inode *inode, struct file *file)
{
    return single_open(file, show_ram_data, NULL);
}

static struct file_operations fops =
{
    .owner = THIS_MODULE,
    .open = open_file_proc,
    .release = single_release,
    .read = seq_read,
    .llseek = seq_llseek,
    .write = write_file_proc
};
```

Por último en la función `show_cpu_data` se hace una llamada al método `for_each_process()` que es el que lee la información del cpu y la guarda en el puntero del struct `task_struct` y a la vez permite recorrer cada proceso obtenido por el método anterior.

Dentro del mismo se utiliza el método `get_task_struct()` para obtener la información individual, y por último se utiliza el `get_mm_rss()` para obtener los datos de la memoria adicionales utilizados en cada proceso.

```

static int show_cpu_data(struct seq_file *m, void *v)
{
    #define K(x) ((x) << (PAGE_SHIFT - 10))
    si_meminfo(&si);

    seq_printf(m, "[");

    countCOMMA = 0;
    for_each_process(task_list)
    {
        unsigned long rss;
        get_task_struct(task_list);

        if(countCOMMA == 0)
        {
            seq_printf(m, "\t{");
        } else
        {
            seq_printf(m, ",\t{");
        }
        seq_printf(m, "\t\t\"PID\": %d,", task_list->pid);
        seq_printf(m, "\t\t\"NOMBRE\": \"%s\",", task_list->comm);
        seq_printf(m, "\t\t\"UID\": %d,", __kuid_val(task_list->real_cred->uid));
        seq_printf(m, "\t\t\"ESTADO\": %ld,", task_list->state);
        if(countCOMMA == 0)
            seq_printf(m, "\t\t\"MM\": {\n");
        if(task_list->mm)
        {
            rss = get_mm_rss(task_list->mm) << PAGE_SHIFT;
            seq_printf(m, "\t\t\t\"RAM\": %lu,", (rss/1024)*100/K(si.totalram));
            seq_printf(m, "\t\t\t\"RAM_BYTES\": %lu,", rss/1024);
        } else
        {
            seq_printf(m, "\t\t\t\"RAM\":0,");
            seq_printf(m, "\t\t\t\"RAM_BYTES\":0,");
        }

        seq_printf(m, "\t\t\"HIJOS\": [");

        countCOMMA2 = 0;
        list_for_each(lista_hijos, &(task_list->children))
        {

```

Comandos en consola:

Con el objetivo de compilar el código escrito en C y que el objeto .ko sea montado en el kernel se utiliza un archivo **Make** el cual contiene las instrucciones y/o comandos necesarios para automatizar este proceso. Además, de manera personal, se agregaron más comandos los cuales ayudan a montar y desmontar el módulo en cuestión de una forma más rápida y simple.

```
cpu_module > Makefile
1   obj-m := cpu_201603168.o
2
3   KDIR := /lib/modules/$(shell uname -r)/build
4
5   all:
6       $(MAKE) -C $(KDIR) M=$(shell pwd)
7
8   clean:
9       $(MAKE) -C $(KDIR) M=$(shell pwd) clean
10
11  up:
12      @echo "NOMBRE: DOUGLAS OMAR ARREOLA MARTINEZ"
13      @sudo insmod cpu_201603168.ko
14
15  down:
16      @echo "DICIEMBRE 2021"
17      @sudo rmmod cpu_201603168.ko
18 |
```

- **all:** compila el código en C y prepara los código objeto en la librería indicada en la variable **KDIR**.
- **make clean:** limpia los archivos generados con el comando make indicados en el directorio **KDIR**.
- **make up:** ejecuta el comando insmod con el que se monta el módulo en el kernel de la distribución de linux. También escribe en consola la frase “NOMBRE: DOUGLAS OMAR ARREOLA MARTINEZ”.
- **make down:** ejecuta el comando rmmod con el que se desmonta el módulo del kernel. También escribe en la consola la frase “DICIEMBRE 2021”.

Con el comando **dmesg** se puede ver los mensajes escritos por los módulos en el buffer de mensajes del núcleo. Al montar y desmontar el módulo **cpu_201603168** se pueden ver los siguientes mensajes:

```
douglas@dell: ~
File Edit View Search Terminal Help
douglas@dell:~$ dmesg
[91195.912977] DOUGLAS OMAR ARREOLA MARTINEZ
[91198.984535] DICIEMBRE 2021
douglas@dell:~$
```

SISTEMA OPERATIVO

El sistema operativo en el que se llevó a cabo la realización del proyecto fue Ubuntu 18.04 LTS de 64 bits, con las siguientes características:

